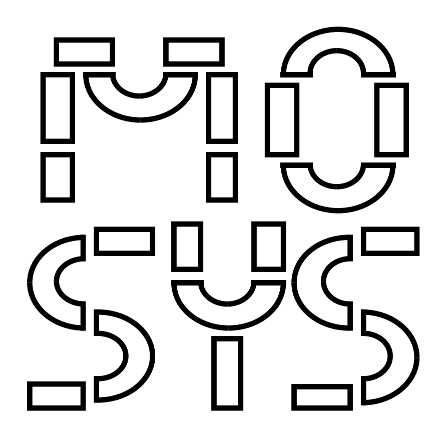
Universität Bielefeld
Technische Fakultät
Postfach 10 01 31
D-33501 Bielefeld



Evaluation hybrider Expertensystemtools

Sonja Schlegelmilch, Barbara Heller, Thomas Linke,
Josef Meyer-Fujara

Reportreihe Modularisierung wissensbasierter Systeme **Nr. 13**
Arbeitsgruppe Wissensbasierte Systeme (Künstliche Intelligenz) **März 1993**

Inhaltsverzeichnis

1. Einleitung

1.1. Motivation für die vorliegende Studie

Im Projekt HYPERCON (HYPERtension CONsultation) – einem Leitvorhaben der Arbeitsgruppe "Wissensbasierte Systeme/Künstliche Intelligenz" – geht es um die Erstellung einer umfangreichen modularen Wissensbasis zur Hypertonie-Konsultation[1]. Hieran sollen Grundlagen zur Konzeptualisierung und Strukturierung des Wissens schwer überschaubarer Wissensdomänen erforscht werden, welche durch neuartige Konzepte der wissensbezogenen Modularisierung erreicht werden soll.

Im angestrebten Einsatzszenario für das entwickelte System wird die Wissensbasis interaktiv von sachkundigen Benutzern (Ärzten) konsultiert. Ihnen soll problem- und fallbezogen Wissen bereitgestellt werden, wobei diese für die Entscheidungen und Interventionen selbst verantwortlich bleiben. Die Arbeiten sollen an die im Vorläuferprojekt HYPERTON geleisteten Vorarbeiten anschließen und die dabei gewonnenen Erfahrungen aufgreifen.

Die informatischen Aufgaben betreffen vor allem die Fortentwicklung und Neukonzipierung von Methoden der Wissensmodellierung im Hinblick auf die Modularisierung durch partitionierte Wissensbasen. Das relevante Domänenwissen wird im wesentlichen aus Expertenbefragungen akquiriert, und mit dem Ziel, dieses für die Entscheidungsunterstützung in der Diagnostik und Therapie zu operationalisieren. Wegen der vielfältigen Ursachen der Hypertonie und ihrer Wechselwirkungen mit anderen Erkrankungen, z.B. des Stoffwechsels (etwa Diabetes), muß eine offene, erweiterbare Form der Wissensdarstellung gefunden werden. Die Annäherung an die breite Wissensdomäne soll nach Leitprinzipien der domänenorientierten Wissensstrukturierung erfolgen, auf die sich Überlegungen zu einer modularen, erweiterbaren Wissensbasiskonzeption gründen. Dabei soll insbesondere auch die Verwendung unterschiedlicher Wissensarten und die von Ebenen unterschiedlicher Detaillierung (Granularität) berücksichtigt werden.

Gemäß grundsätzlicher Entscheidungen in der Definitionsphase des Projekts HYPERCON soll die Implementierung des geplanten Systems mit Hilfe eines markteingeführten Werkzeugs durchgeführt werden. Dieser Entscheidung liegt u. a. die Ansicht zugrunde, daß damit eine höhere Zuverlässigkeit des Systems erreichbar ist. Des weiteren sind die meisten der high-end

[1] Eine zusammengefasste Projektbeschreibung ist als MOSYS-Report Nr. 10 verfügbar

Werkzeuge mit den wesentlichen der insgesamt benötigten Repräsentationsformalismen und Inferenzmechanismen ausgestattet.
Der Aufwand, eine Repräsentations- oder Inferenzkomponente selbständig zu programmieren, erscheint im Hinblick auf die Projektintention nicht vertretbar. Die Komplexität der zu modellierenden Domäne erfordert in jedem Fall den Einsatz eines hybriden Entwicklungswerkzeugs, das über mehrere Formalismen verfügt.

1.2. Evaluierte Werkzeuge und Evaluationskriterien

Um zu einer fundierten Entscheidung über das geeignetste Werkzeug zu gelangen, wurden mehrere markteingeführte hybride Software-Werkzeuge zur Erstellung wissensbasierter Systeme im Hinblick auf ihre Eignung für das Projekt vergleichend evaluiert.

Der Arbeitsgruppe standen vor dem Projektbeginn mehrere Werkzeuge zur Verfügung: Knowledge Craft, KEE, Nexpert Object, Goldworks und Kappa-PC. Diese wurden einerseits für die Lehre in Übungen und Praktika zum Thema "Expertensysteme" eingesetzt. Andererseits war in der Arbeitsgruppe durch Versuchen, die mit den o. g. Werkzeugen realisiert wurden, eine gewisse Benchmarking-Erfahrung vorhanden, die zu einem detaillierten Fragenkatalog führte.

Darüberhinaus besitzt das Projektteam umfassende Erfahrungen in der Implementierung von größeren Wissensbasen mit Expertensystemwerkzeugen aus früheren Projekten.

Zu Beginn des Projektes wurden folgende zu evaluierende Werkzeuge ausgewählt:

- Knowledge Craft, KEE (aus dem vorhandenen Bestand),
- ProKappa, babylon, ROCK (für 2,5 Monate leihweise beschafft und zu Testzwecken installiert).

Die o. g. zu evaluierenden Werkzeuge wurden auf SUN Sparc Grafik-Workstations mit 32 MB RAM und mehr als 120 MB Swapspace installiert. Als Betriebssystem lag UNIX Version 4.1.2 bzw. 4.1.3 zugrunde.

Die bereits vorhandenen, jedoch nicht zur Evaluation herangezogenen Werkzeuge zeichneten sich durch eine mindere Flexibilität und eine geringere Repräsentationsmächtigkeit aus und waren für die Anforderung – die Erstellung großer modularer Wissensbasen im Team – nicht ausgelegt.
Im folgenden werden einige Evaluationskriterien hervorgehoben, die vor dem Hintergrund der spezifischen Projektausrichtung wichtiger erscheinen, als dies bei einer allgemeinen Bewertung

von Expertensystemwerkzeugen der Fall wäre. Sie lassen sich zusammenfassen unter den Aspekten:

1. Adäquatheit für die zu modellierende Wissensdomäne,
2. systemtechnische Voraussetzungen für die Modularisierung,
3. Art, Umfang und Handhabbarkeit der Darstellungsmittel.

Die Komplexität der Wissensdomäne erfordert eine Betrachtung aus verschiedenen Sichten. Deshalb müssen Framestrukturen bzgl. der taxonomischen Repräsentation als Netz – und nicht nur als Baumhierarchie – organisiert werden können. Ferner sollten Frames in ihren Slots Verweise auf Objekte erlauben und in ihrer Slotstruktur so flexibel sein, daß Listen untypisierter Datenobjekte möglich sind. Klassen- und Instanzenslots müssen getrennt verarbeitbar sein. Die Verkettungs- und Konfliktlösungsstrategie bei der Auswertung von Regeln sollte der Entwickler selbst definieren können.

1.3. Projektbezogene und systemtechnische Aspekte

Die Domäne wird weitgehend durch die Diagnoseproblematik charakterisiert. Daraus folgt, daß die gleichzeitige Elaborierung bzw. der Vergleich verschiedener Hypothesen eine wichtige Rolle spielt. Technisch gesehen müssen deshalb Ergebnisse verschiedener Annahmen nebeneinander verwaltet werden können. Ebenso sollte sich die konkrete Abhängigkeit der Ergebnisse von jeweils zugrundeliegenden Annahmen verfolgen lassen. Die möglichen Inkonsistenzen, die Folgerungen aus unterschiedlichen Annahmen untereinander aufweisen können, müssen vom System toleriert werden; eine technische Möglichkeit hierzu ist die Verwaltung hypothetischer Welten. Die Qualität einer solchen Verwaltung hängt unter anderem davon ab, ob sie nur unterschiedliche Slotwerte oder auch z.B. unterschiedliche Instanzenmengen einer Klasse erlaubt, ferner davon, inwieweit Redundanzen erkannt und durch effiziente Speicherung aufgefangen werden. Ähnliche Anforderungen ergeben sich aus der Verwendung von Modellen unterschiedlicher Detaillierung. Hinzu kommt, daß Beziehungen zwischen Objekten, die verschiedenen Modellen zugeordnet sind, beschrieben werden können müssen. Dies scheint technisch am ehesten durch die Beschreibung zwischen den Objekten bestehender Relationen möglich, wobei der Knowledge Engineer die Vererbung von Eigenschaften relationsspezifisch festlegen können sollte.

Da es sich bei der Hypertonie um eine chronische Krankheit handelt, die oft langsam progrediert, werden Patienten auch in größeren Abständen untersucht. Die Bewertung der zeitlichen Entwicklung der Patientendaten ist für die Diagnose, Prognose und Therapie dieser

Erkrankung notwendig. Aus diesem Grund sollte das System eine Zeitverwaltung besitzen, mit deren Hilfe Folgerungen aus individuellen Zeitverläufen gezogen werden können.
Im Bereich der Hypertonie stehen umfangreiche Falldatenbanken zur Verfügung, die möglichst genutzt werden sollten, um den Benutzer des Systems durch Fallvergleiche und evtl. deren Aufbereitung unter epidemiologischen Gesichtspunkten zu unterstützen. Darüber hinaus sollte das angestrebte System mit Hilfe dieser Datenbanken validiert werden können, mindestens jedoch durch einige aus dem Bestand ausgewählte Testfälle. Für das Werkzeug ergibt sich daraus die Anforderung, den effizienten Anschluß einer Datenbank zu unterstützen.

Um eine Modularisierung von Wissensbasen zu erreichen, ist es notwendig, die Gesamtwissenbasis in Teilwissensbasen aufzuteilen, die im Verlauf einer Systemkonsultation quasiparallel benutzt werden. Dabei sollten Teilwissensbasen nur für den Zeitraum, in dem sie für den Konsultationsprozeß benötigt werden, geladen sein. Dies erfordert aus systemtechnischer Sicht eine dynamische Wissensbasenverwaltung. Dementsprechend sollte die Regelauswertung leicht auf bestimmte Teile der gesamten Regelmenge eingeschränkt werden können.

Das Ziel der parallelen Implementation durch mehrere Projektmitarbeiter erfordert Freiheit in der Reihenfolge des Eintrags von Wissenselementen und des Ladens von Teilen der Wissensbasis. Um eine möglichst große Unabhängigkeit der Teammitglieder bei der Entwicklung zu erreichen – nur sie läßt die Realisierung umfangreicher Projekte zu –, muß das Werkzeug Möglichkeiten zum Umgang mit Namenskonflikten zwischen verschiedenen Teilwissensbasen bereitstellen.

Eine Versionsverwaltung, die eine Kennzeichnung von Objekten wie Frames und Regeln mit Urheber und Zeitpunkt von Entstehung und Änderungen unterstützt, ist für eine übersichtliche Entwicklung im Team unabdingbar.

Allgemein kann nicht erwartet werden, daß ein System alle Repräsentationsformalismen und Kontrollstrategien bereitstellt, die letzlich benötigt werden. Auch ist zu erwarten, daß sich der Bedarf an Formalismen und Strategien teilweise erst während der prototypischen Realisierung von Systemkomponenten herausstellt. Eine wesentliche Forderung an das Werkzeug ist deshalb seine Offenheit für Ergänzungen durch die Entwickler. Das bedeutet insbesondere, aus den vordefinierten Programmierkonstrukten heraus auf speziell programmierte Funktionen zugreifen zu können – technisch gesehen also die komfortable Verwendbarkeit der vollen unterliegenden Programmiersprache.

Speziell ist hier an die Anbindung neuronaler Komponenten und an die Programmierung von Constraints zu denken, die selten als Standardformalismus angeboten werden. Auch die

Steuerung und Protokollierung des Inferenzprozesses werden vermutlich auf die Programmiersprache zurückgreifen müssen.
Das zu realisierende System ist derart umfangreich und komplex, daß in Anbetracht des engen Zeitrahmens Risiken bzgl. der nachfolgend aufgeführten Punkte im Vorfeld des Projektes zu vermeiden sind:
- mangelnde Information über die Fähigkeiten des Werkzeugs,
- suboptimale Installation,
- ungenügende Unterstützung des Programmdebuggings,
- nicht aufgedeckte oder verzögert behobene Fehler des Werkzeugs.

Deshalb sind die Qualität von Dokumentation und Support und die Zuverlässigkeit des Werkzeugs bedeutende Evaluierungskriterien. Um ein redundanzarmes und damit in der Prototyping- und Testphase effizientes System zu erhalten, muß das Werkzeug konfigurierbar sein, d.h. nicht benötigte Funktionalität muß sich effektiv ausblenden lassen, so daß ein bedarfsgerechtes Subsystem generiert werden kann.

2. Evaluation von babylon

babylon ist ein hybrides, auf LISP basierendes Programmierwerkzeug, das in erster Linie für industrielle Anwendungen der künstlichen Intelligenz vorgesehen ist. Grundlage der Entwicklung war die von der Forschungsgruppe Expertensysteme der GMD entwickelte KI-Werkbank babylon, die grundlegende Repräsentationsformalismen umfaßt.

2.1. Produktbeschreibung

Die Babylon-Version der GMD wurde zunächst nur ohne graphische Etnwicklungsoberfläche angeboten. Dieses Defizit ist bei der Überarbeitung dieser Version durch die VW-Gedas GmbH zu einem graphisch unterstützten Expertensystem-Tools für Workstations behoben worden. Dabei entstand allerdings ein stark verändertes Produkt: babylon. Syntax und Semantik der Basissprache wurden modifiziert und zum Teil eingeschränkt. Dennoch blieb die Offenheit des Systems soweit erhalten, daß auch weiterhin Modifikationen der Meta-Repräsentationsebene möglich sind.

babylon bietet die Möglichkeit, eine auf Fenstern, Ikonen und Menüs basierende Endbenutzerschnittstelle zu schaffen, die sich auch zum Testen des entwickelten Systems in der endgültigen Benutzerumgebung eignet.

Die babylon zugrundeliegende Programmiersprache ist Lisp; dies stellt auch die Basis der Repräsentation dar. babylon ist auf Unix-Workstations, IBM-Mainframes unter VM/MVS und auf Macintosh-Rechnern verfügbar. Der Version für SUN-Workstations liegt Allegro Common Lisp (ACL) zugrunde, eine Erweiterung von Common Lisp. Bei der Entwicklung von babylon wurde speziell auf die Verwendung von Standards wie OSF/Motif, X-Windows und SQL Wert gelegt.

2.2. Produktarchitektur

babylon besteht aus den Komponenten babylon-Entwicklungsumgebung, babylon-Kern und babylon-Kommunikationsmodule (vgl. Abb. 1).

Der **babylon-Kern** beinhaltet die Konstrukte zur Erstellung einer Wissensbasis; sowohl deklaratives Wissen (Frames) als auch prozedurales Wissen (Regeln, Constraints, Tasks) werden unterstützt. Auf die verschiedenen Wissensrepräsentationsformalismen kann mit Hilfe der Schnittstelle *babylon-Query-Language* (BQL) in einheitlicher Weise zugegriffen werden.

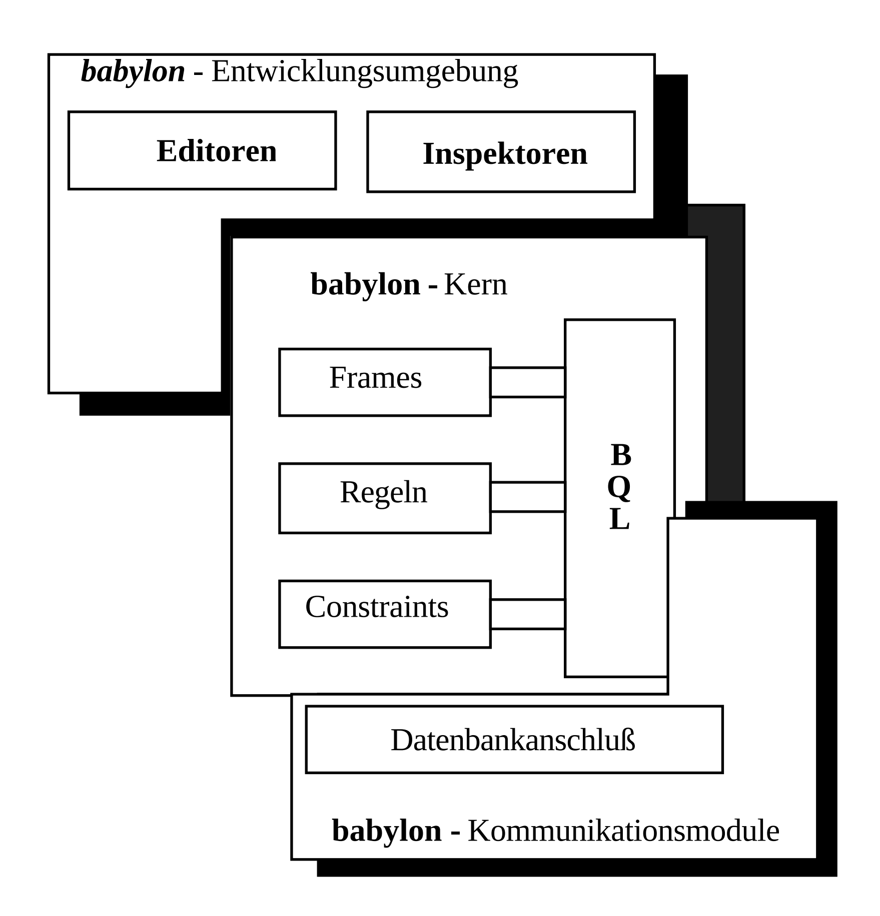


Abb. 1: Die Systemarchitektur von babylon (nach: *babylon* Benutzerhandbuch, VW-Gedas GmbH)

Die **babylon-Entwicklungsumgebung** beinhaltet für alle Repräsentationskonstrukte spezielle Editoren. Während diese Editoren jeweils ein zu bearbeitendes Konstrukt fokussieren, gewähren die zu den Inspektoren gehörigen Browser einen Überblick über die Gesamtheit der Konstrukte eines Typs und über Beziehungen zwischen ihnen. Zu den Inspektoren gehören weiter Protokoll-, Statistik- und Tracekomponenten. Zur Entwicklungsumgebung sind wohl auch die Systemwerkzeuge (Hilfe-, Fehler- und das Editor-ähnliche Kommandowerkzeug) zu rechnen sowie die zur Verfügung stehende Schnittstelle für den Wissensingenieur ("Wisssensingenieurschnittstelle") auf der Lisp-Ebene und die Funktionen zur Erzeugung von Wissensingenieurschnittstellen ("Programmierschnittstelle") und Endbenutzerschnittstellen.

Die **babylon-Kommunikationsmodule** bieten mit dem Datenbankanschluß die Möglichkeit, SQL-Anfragen an eine externe Datenbank zu richten. In den vorliegenden Handbüchern werden darüber hinaus aber keine weiteren Angaben zu diesem Punkt gemacht. Zu den Kommunikationsmodulen sollte man auch den Standard-IO-Handler der Wissensbasis (Wissensbasismanager) rechnen.

2.3. Wissensrepräsentationsformalismen

babylon bietet als folgenden Wissensrepräsentationsformalismen[2] Frames, Regeln, Constraints und Tasks an.

Wissen über Objektklassen wird in hierarchisch geordneten **Frames**, die durch Eigenschaften und zugeordnete Prozeduren (sog. Behaviors) beschrieben werden.

Konkrete Objekte, die diesen Klassen entstammen, werden als **Instanzen** bezeichnet. Nur statisch definierte Instanzen können über ihren Namen angesprochen werden. Dynamische Instanzen, die erst zum Konsultationszeitpunkt der Wissensbasis erstellt werden, sind möglich; diese müssen aber an Symbole oder Variable gebunden werden, um zugreifbar zu sein.

Frames und Instanzen sind durch zwei Arten von Relationen beschreibbar: **Merkmale** (= general relations, sie beschreiben Eigenschaften in einem bestimmten Wertebereich) und sogenannte **Bestandteile** (= part relations, sie referieren auf andere Frames oder Instanzen, diese sind aber nicht notwendigerweise Aggregatkomponenten). Zur Weitergabe der Werte dieser Slots[3] an Subframes (in der Hierarchie tiefer stehende Frames) sind sowohl einfache wie auch multiple Vererbung vorgesehen; letztere erfolgt stets additiv. Eine Modifikation dieser vordefinierten Vererbungsmechanismen ist nicht möglich, auch gibt es keine nicht-vererbbaren Slots. Defaults für Slotwerte können an jeder Stelle der Frame-Hierarchie gesetzt werden, sie werden stets vererbt, können aber in Subframes überschrieben werden. Die Überschreibungsmöglichkeit kann allerdings durch explizite Anweisung generell oder auch speziell für die Initialisierung aufgehoben werden. Wertrestriktionen sind nur sehr eingeschränkt möglich (Anzahl, Lisp-Typen und Frames).

[2]wir verwenden im folgenden die Begriffe Faktenwissen, Relation und Bestandteil in dem ungewöhnlich engen/weiten Sinne, der ihnen im Benutzerhandbuch gegeben wird.

[3]Wir fassen die Begriffe Merkmale und Bestandteile unter dem üblichen Ausdruck *Slot* zusammen

Bei jedem Slot können Wertanfragen/Zuweisungen sogenannte **Behaviors** auslösen. Das sind Prozeduren, die über Seiteneffekte andere Werte beeinflussen können. Die Identifikation der zu einem Frame gehörigen Behaviors geschieht durch Prüfung des Argumenttyps. Dadurch können für spezielle Frames leicht spezifische Behaviors definiert werden. Das Behavior-Konzept wird noch mächtiger durch die Möglichkeit, den zentralen (*:primary*-)Behaviors vor- oder nachgeschaltete (*:before*/*:after*) Behaviors zu verwenden, die die taxonomische Hierarchie auf- bzw. absteigend nacheinander aufgerufen werden.

Im angebotenen **Regelkalkül** werden Regeln stets zu Regelmengen zusammengefaßt. Für Regelmengen muß in sogenannten *Regelmengen-Tasks* jeweils ein Abarbeitungsmodus und eine Konfliktlösungsstrategie angegeben werden. Regelmengen-Tasks können Parameter spezifizieren, die beim Aufruf von Regeln mit aktuellen Werten ausgefüllt werden. In ihrem sogenannten Aktionsteil dürfen beliebige *babylon-lisp*-Ausdrücke auftreten; insbesondere können die zur Regelmenge gehörenden Regeln sowie andere Tasks angestoßen werden. In den Regelprämissen sind außer BQL-Prädikationen und damit Variablenbindungen auch Fragen an den Benutzer möglich. Tasks können auch durch die Konklusion von *Einzelregeln* aktiviert werden.

Jeder Regel kann innerhalb ihrer Regelmenge eine Priorität (*:priority*) zugeordnet werden, die später Einfluß auf die Auswertungsreihenfolge hat. Eine Hierarchisierung der Regelmengen-Tasks untereinander ist durch die Angabe von Subtasks möglich; es findet dabei jedoch keine Eigenschaftsvererbung statt.

Constraints sind Kontrollinstanzen, mit denen wechselseitige Abhängigkeiten zwischen Slot-Werten ausgedrückt werden. Sie können Einschränkungen in der Ausprägung einzelner Objektattribute oder in der Ausprägung von Kombinationen mehrerer Objektattribute erzwingen. Constraints werden als Instanzen sogenannter *Constraint-Patterns* eingerichtet. Diese beinhalten Muster-Restriktionen bzw. Rechenvorschriften. Die einzelnen Constraints selbst geben dann an, welche konkreten Werte (d.h. welche Slotwerte welcher Instanzen) in der Art, die im Pattern beschrieben ist, verknüpft werden sollen. Ein *For-each*-Konstrukt ermöglicht es, gleichartige Constraints für alle Elemente einer Menge von Objekten mit einer einzigen Anweisung zu instantiieren.

Auf das in Frames und Instanzen abgelegte Wissen kann mit Hilfe der Babylon-Query-Language (BQL) zugegriffen werden. Typische BQL-Ausdrücke enthalten Operatoren wie *ask*, *tell*, *retell* oder *untell*. Es werden Prädikationen über Wissensbasisinhalte (Frame-Instanz und Instanz-Wert-Beziehungen) unterstützt, die mit den logischen Standardoperatoren verknüpft werden können. Darauf aufbauend stellt BQL komfortable Möglichkeiten der Variablenbindung

zur Verfügung, die z.B. die automatische Iteration über alle Belegungen erlauben, die die Prädikation erfüllen. Die bei der Iteration auszuführenden Operationen werden dabei in einer sogenannten *continuation* in der Art eines Lisp-Funktionsrumpfs aufgeführt. Die continuation erlaubt zwar (nach Handbuchangaben) den Zugriff auf die gesamte Syntax von Common Lisp; es wird aber geraten, sich auf den Sprachumfang von *babylon-lisp* zu beschränken. Im Test ist der Versuch, andere Lisp-Ausdrücke einzubinden, dann auch gescheitert.

Zu jedem *babylon*-Konstrukt sind zwei Erläuterungskomponenten vorgesehen: Der **Dokumentationseintrag** für konstruktspezifische Informationen des Wissensingenieurs für spätere Wartungsarbeiten und der **Erklärungseintrag**. Dieser soll spezifische Informationen für den Endbenutzer zum Verstehen und Einordnen der Konstrukte beinhalten. Unterschieden wird zwischen drei Informationstypen: Beschreibungen von Wissensbasiskonstrukten (*:description*/*:specific-description*), Wertbestimmungshilfen zur Bestimmung aktueller Attributwerte (*:determination*/*:specific-determination*) und Ausführungsbeschreibungen, die auf die Parameter der konkreten Anwendung zugreifen, um die jeweilige Auswirkung zu beschreiben (*:effect-description*). Darüberhinaus können auch eigene Informationstypen definiert werden und Konstrukte der Wissensbasis als nicht erklärbar (*:not-explainable*) gekennzeichnet werden.

2.4. Inferenzmechanismen und Kontrollfunktionen

Bei Frames werden bei Lese- / Schreibzugriffen auf Slots dämonenartig **Behaviors** ausgelöst. Welche Zugriffe ein Behavior auslösen, wird bei der Merkmalsdeklaration in der Framedefinition festgelegt (*:read-dependents, :write-dependents, :if-undetermined*). Für die Mitteilung von Schreib-/ Lesezugriffen sind spezielle Behaviors vorgesehen (*notify-read-dependent, notify-write-dependent*); die Adressaten der Mitteilung können bei den Slots explizit angegeben werden.

Für **Regelmengen** stehen drei Abarbeitungsmodi zur Auswahl: Vorwärts- oder Rückwärtsverkettung sowie sequentielle Abarbeitung der Regeln.
Bei der *Vorwärtsverkettung* werden die Regeln in der Sortierreihenfolge getestet und Regeln mit gültiger Instantiierung in eine Konfliktmenge gestellt. Die Sortierreihenfolge ergibt sich dabei aus der Priorität der Regeln oder bei gleicher Priorität aus der Aufschreibreihenfolge. Aus der Konfliktmenge wird dann nach einer der drei Konfliktlösungsstrategien *:first-match, :newest-match* oder *:oldest-match* (andere sind nicht möglich) eine einzige Regel ausgewählt und ausgewertet. Danach beginnt der Prozeß von vorne, wobei dieselbe Regel aber nicht noch einmal auf denselben Daten feuert. Beendet werden kann dieser Vorgang auf drei Arten: *do-*

exhaustive (solange, bis keine Regel mehr auswertbar ist), *do-while* (solange, wie eine bestimmte Bedingung vor der Regelauswahl gilt) oder *do-until* (solange, bis eine bestimmte Bedingung am Ende einer Regelausführung erfüllt ist).
Innerhalb von *rückwärtsverketteten Regelmengen* gibt es nur die Regelauswertungstaktik *verify*. Diese versucht, eine explizit anzugebende Prädikation zu verifizieren. Die Ableitungsstrategie ist hierbei *depth-first*, *left-to-right*, wobei die Sortierreihenfolge entscheidend bei der Auswahl der zur Etablierung der Prädikation herangezogenen Regeln ist.

Beim *sequentiellen Abarbeitungsmodus* werden die Regeln einer Regelmenge stets in derselben Reihenfolge nacheinander betrachtet. Es werden dabei verschiedene Auswertungstaktiken angeboten: *do-one* (prüft die Regeln in der Sortierreihenfolge und wertet nur die erste ausführbare Regel aus), *do-all* (alle Regeln werden geprüft und ggf. ausgewertet), *do-while* (Testen der Regeln in der Sortierreihenfolge, bis eine angegebene Bedingung nicht mehr erfüllt ist) oder *do-until* (Testen der Regeln in der Sortierreihenfolge, bis eine angegebene Bedingung erfüllt ist). Dabei können Regeln wiederholt mit der gleichen Instantiierung angewendet werden.

In einem **Constraint-Pattern** wird eine Reihe von sogenannten Regeln angegeben, die das Verhalten für bestimmte Situationen, die in einem Prämissenteil beschrieben werden, festgelegen. Das in der Konklusion der Regel ausgedrückte Verhalten kann zum einen darin bestehen, eine Situation als :INCONSISTENT kenntlich zu machen (wenn die aktuellen Parameterwerte des Patterns eine unzulässige Kombination untereinander und/oder mit der Wissensbasis darstellen), zum andern darin, Variablen Werte zuzuweisen, die typischerweise aus den Werten anderer im Constraint-Pattern vorkommender Variablen berechnet werden. Zum Prämissenteil einer Regel gehört obligatorisch eine *known*-Klausel, die beschreibt, für welche Variable mindestens konkrete Werte bekannt sein müssen. Zusätzlich können optional weitere Kriterien (Prädikationen) unter dem Keyword *and-if* aufgeführt werden, die den Reaktionsraum des Constraints stärker einschränken. Bei der Bearbeitung eines **Constraints** prüft babylon nacheinander für alle Regeln des zugehörigen Patterns, ob die in ihrer Prämisse beschriebene Situation vorliegt. Falls ja, wird diese Regel, und nur diese, ausgeführt.

Mit sogenannten **Tasks** ist ein Mittel zur strukturierten Steuerung der Abarbeitung spezifischer Aufgaben in der Wissensbasis gegeben. Tasks beinhalten eine Kette von BQL-Ausdrücken, die bei Aufruf der Task sequentiell abgearbeitet werden. Dabei kann beliebig mit den Tell-and-Ask-Konstrukten von BQL auf die Elemente der Wissensbasis zugegriffen werden; auch der Start weiterer Tasks ist möglich; ihnen können dabei Parameter übergeben werden. Die Konsultation einer Wissensbasis beginnt stets über Tasks, hierfür muß eine spezielle Task als Starttask ausgewiesen werden.

2.5. Entwicklungsunterstützung

Das Tool ermöglicht mit seiner graphischen Programmieroberfläche ein recht geschwindes Eingeben von Konstrukten, obwohl einzelne Werkzeuge beim Hochfahren des Werkzeugs erst verzögert (nach bis zu 30 sec) erscheinen. Speziell sind folgende graphische **Editoren** verfügbar: Frame-Editor, Instanz-Editor, Behavior-Editor, Regel-Editor, Regelmengen-Editor, Task-Editor, Constraint-Pattern-Editor, Constraint-Editor, graphischer Editor zur Instanzen-/Frame-Generierung und schließlich ein Konfigurator zur Dokumentation und Interface-Konfiguration für die Wissensbasis. Mit den einheitlich gestalteten Editoren ist eine Veränderung aller statischen Komponenten der Wissensbasis möglich.

Vorhandene **Inspektoren** sind: alphanumerischer Browser, graphischer Browser, Protokollkomponente zur Ablaufverfolgung, Statistikwerkzeug und ein Tracer. Auch diese haben ein einheitliches Design. Ein Überblick über (im Test nur kleine) statische Wissensbasisinhalte ist leicht zu erhalten, Veränderungen der dynamischen Wissensbasisinhalte lassen sich aber mit Hilfe des graphischen Browsers nur umständlich verfolgen. Die Bedienung von Protokollkomponente und Tracer benötigt einige Einarbeitung; Protokolle können in unterschiedlicher Detaillierung aufgezeichnet und auch später mit Statistikwerkzeug und Tracer ausgewertet werden.

Ein **Hilfe-Werkzeug** ist zwar vorhanden, es bezieht sich aber meist auf die Bedienung der Oberfläche, es werden keine Hilfestellungen zur Wissensdarstellung gegeben.

In einem **Notizwerkzeug** können gegebene Hilfeleistungen leicht festgehalten werden, auch eigene Einträge sind möglich. Leider ist die Fenstergröße dieser Werkzeuge nicht an die dargestellten Inhalte angepaßt, so daß ein häufiges Fensterhandling nötig ist.

Das **Fehlerwerkzeug** weist große Unzulänglichkeiten auf: Inkonsistenzen in der Wissensbasis oder bei der Evaluation werden zwar entdeckt, aber nur sehr schlecht oder gar nicht kommentiert, geschweige denn mit Korrekturvorschlägen versehen. Häufig werden lediglich die Fehlermeldungen von ACL unkommentiert weitergegeben. Der dann auswählbare Debugger zwingt zum Verlassen der graphischen Oberfläche und zum Debuggen im Lisp-Listener.

Der von allen Editoren und Browsern erreichbare **File-** und **Editor-Manager** ist nach kurzer Einarbeitung recht gut zu bedienen. Leider können Defaultwerte für Standard-Directories nicht benutzerspezifisch gespeichert werden, so daß bei jeder Sitzung alle Pfade neu eingegeben werden müssen.

Bei ringförmig referenzierenden *part relations* kann es vorkommen, daß beim Speichern und erneuten Laden der Wissensbasis ein Fehler gemeldet wurde, der bei der interaktiven Erstellung der Wissensbasis nicht aufgetreten war. Dies ist auf eine strikte Reihenfolgeabhängigkeit beim Laden und Evaluieren von Wissensbasen zurückzuführen: Stets darf nur auf bereits bekannte Konstrukte der Wissensbasis referiert werden. Beim Editieren von Quelltexte sollte man deshalb darauf achten, daß Einschränkungen für die Reihenfolge der Konstrukttypen bestehen. Ein Vorschlag zu einer Standardreihenfolge wird im Benutzerhandbuch gegeben.

Obwohl verschiedene Wissenbasen parallel im Speicher gehalten werden können, sind doch Kommunikationen zwischen unterschiedlichen Wissensbasen nicht möglich. Eine Modularisierung wird auf dieser Ebene nur soweit unterstützt, als Teile derselben Wissensbasis in unterschiedlichen Source-Code-Files gespeichert sein können, zum Konsultationszeitpunkt muß aber stets der gesamte referenzierte Wissensvorrat geladen und evaluiert sein. Die Zusammenfassung von Regeln zu Regelmengen stellt hingegen eine sinnvolle Modularisierungsmöglichkeit dar.

2.6. Dokumentation und Support

Von VW-Gedas werden ein Benutzerhandbuch und ein Referenzhandbuch bereitgestellt. Im Benutzerhandbuch finden sich allgemeine Grundlagen der Wissensrepräsentation (speziell auf babylon-Syntax hin ausgerichtet), Erklärungen zur Maus- und Fenstertechnik, zur Werkzeugumgebung und zur Erklärungskomponente, ferner ein Listing der mitgelieferten Beispielwissensbasen. Das Referenzhandbuch beschreibt alle verfügbaren Funktionen in alphabetischer Reihenfolge, allerdings auf unterschiedliche Kapitel verteilt. Sein Index beinhaltet fast ausschließlich Funktionsnamen. Für den Erstbenutzer mit speziellen Fragen ist es deshalb keinerlei Hilfe. Generell erschweren Unsystematik und teilweise Inkonsistenz der Darstellung das Verständnis; Konflikte zwischen den Notationen von Objekt- und Metasprache kommen hinzu. Der Kontext von Beispielen wird häufig nicht ausreichend dargestellt, um sie nachvollziehbar zu machen; teilweise decken sie die Möglichkeiten der Konstrukte nur partiell ab. Insgesamt führen Strukturierung und Gliederung der Handbücher nicht zu einer effizienten Vermittlung von Inhalten.

Außerdem sind die Handbücher ausschließlich auf Syntax- und Semantikbeschreibung hin ausgerichtet; an keiner Stelle wird Hintergrundwissen über das Tool oder dessen Implementierung, insbesondere die der Inferenzmechanismen, vermittelt. Eine Kenntnis derartiger Sachverhalte könnte sicherlich nicht unerheblich zur effizienteren Implementierung beitragen.

Schließlich fehlt eine zusammenhängende Darstellung der Systembedienung.

Der telefonische Support wurde mehrfach in Anspruch genommen; die Auskunftszeiten waren dabei, auch durch Weiterverweise bedingt, teilweise unerwartet groß.

2.7. Zusammenfassende Bewertung

babylon bietet eine Framesprache mittlerer Mächtigkeit, kann aber nicht in vollem Umfang objektorientierte Programmierung anbieten. Eine recht große Einschränkung liegt in der schlechten Verwendbarkeit dynamischer Instanzen. Ähnlich genügt der zur Verfügung stehende Regelkalkül mittleren Ansprüchen. Positiv ist hier die Modularisierungsmöglichkeit durch die Regelmengen hervorzuheben. Die Constraintkomponente stellt eine deutliche Erweiterung der üblichen Repräsentationsformalismen dar; ihre Mächtigkeit läßt sich anhand der Unterlagen nicht zuverlässig einschätzen (Propagierung in Constraintnetzen, instanzübergreifende Constraints). Die Kontrollkonstrukte in BQL und Tasksystem erscheinen komfortabel.

Eine wirkliche Modularisierung durch Aufteilung des Wissens (speziell auch des in der Framehierarchie repräsentierten Objektwissens) auf verschiedene Module, die nach Bedarf getrennt geladen und ausgewertet werden können, deren Ergebnisse aber gemeinsam weiterverarbeitet werden müssen, wird durch die Reihenfolgerestriktionen beim Laden von Wissensbasen und die fehlende Schnittstelle für die Übergabe von Ergebnissen zwischen ihnen verhindert.

Eine Unterstützung von Simulationen wird nicht geboten, ebensowenig die Möglichkeit, Schlußfolgerungen abhängig von ihren Gültigkeitsvoraussetzungen in Welten oder Kontexten zu verwalten. Schließlich wird auch der Umgang mit unsicherem Wissen nicht unterstützt.

Die aktuell zur Verfügung stehende Dokumentation weist erhebliche Schwächen auf.

3. Evaluation von KEE

KEE (Knowledge Engineering Environment) ist ein umfangreiches offenes und hybrides Expertensystemtool, das auf Lisp basiert, welches dem Entwickler eine Vielzahl von KI-Programmierkonstrukten und graphischen Möglichkeiten zur Verfügung stellt. Bei der Bewertung von KEE ist es wichtig, zu berücksichtigen, daß die KEE-Produktfamilie Zusatzmodule beinhaltet, die über das hier dargestellte hinausgehende Möglichkeiten anbieten. So sind ein Simulationsmodul, eine Datenbank- und eine C-Schnittstelle sowie ein Konfigurations- und ein Runtime-Modul erhältlich. Getestet wurde hier allerdings nur die Entwicklungsversion ohne Zusatzmodule.

3.1. Produktbeschreibung

KEE wird von der Firma IntelliCorp Inc. (USA) produziert und wird in Deutschland von der Filiale IntelliCorp GmbH in München vertrieben und unterstützt. KEE ist für folgenden Plattformen erhältlich: Sun-Sparc Familie (Sun OS 4.1.2), HP 700 & 800 (HP-UX 8.07 oder 9.0), HP 300 & 400 (HP-UX 8.07) und IBM RS 6000 (AIX 3.2.2). Nach eigenen Aussagen wird IntelliCorp allerdings zukünftig den Entwicklungsschwerpunkt auf die Produktfamilien Kappa und ProKappa legen und geht von einem Auslaufen der Maintenance von KEE innerhalb der nächsten fünf Jahre aus.

Die von KEE unterstützten Konstrukte sind Frames, objektorientiertes Programmieren (Methoden, Dämonen), Regeln mit Vorwärts- und Rückwärtsverkettung, ein Truth-Maintenance-System und multiple Welten. Zusätzlich werden umfangreiche Farbgrafikmöglichkeiten zur Verfügung gestellt. Unter KEE steht prinzipiell der volle Befehlssatz von Common Lisp zur Verfügung.

Aufgrund der Komplexität der Entwicklungsumgebung von KEE können größere Anwendungen, wie z.B. medizinische Diagnose, nur unter Verwendung eines zusätzlichen Moduls der KEE Produktfamilie in performante Laufzeitsysteme transformiert werden. Darüber hinaus werden weitere Ergänzungsmodule zur KEE-Entwicklungsumgebung angeboten, die je nach Anwendung sinnvolle Erweiterungen darstellen (z.B. Datenbankanschluß).

3.2. Produktarchitektur

KEE basiert auf **Common Lisp** und stellt dessen vollen Befehlssatz sowie einen sehr großen Teil der Funktionen zur Verfügung, aus welchen KEE selbst besteht. Somit können selbstdefinierte Funktionen problemlos eingebunden werden.

Den Kern von KEE bildet das **objektorientierte Programmieren**. Dieses umfaßt eine mächtige Framerepräsentation, Methoden und Dämonen. Mehrere Wissensbasen können problemlos gleichzeitig gehandhabt werden. KEE selbst besteht zu einem großen Teil aus KEE-Wissensbasen. So sind z.B. die Vererbungs- und Inferenzstrategien als Objekte in bestimmten Systemwissensbasen repräsentiert. Diese Systemwissensbasen stehen dem Benutzer zur Modifikation und Ergänzung offen.

Die Produktarchitektur von KEE ist in der folgenden Abbildung dargestellt:

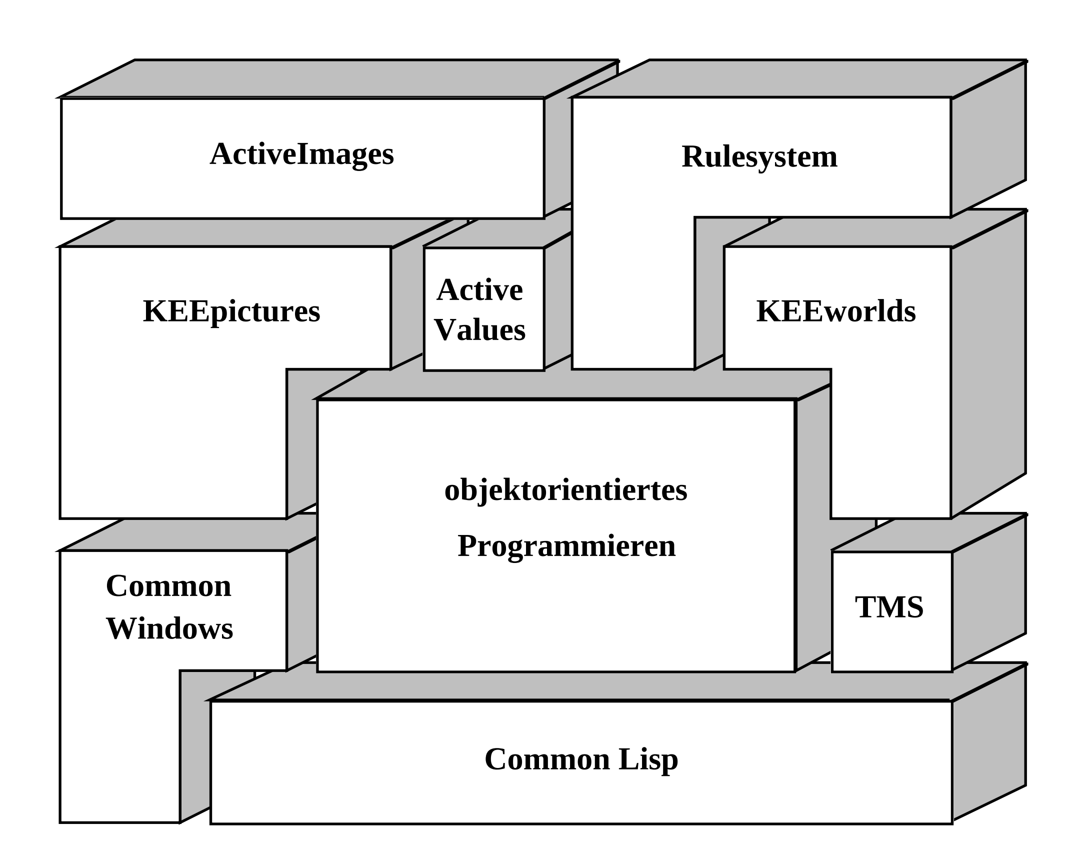


Abb. 2: Die Systemarchitektur von KEE (nach: *KEE User's Guide*, IntelliCorp Inc.)

Das Assumption-based Truth-Maintenance-System (**ATMS**, welches in den Handbüchern unter dem Oberbegriff TMS aufgeführt wird) wird mittels spezieller Regeln (deren Syntax unterscheidet sich von derjenigen der anderen Regeln nur durch ein bestimmtes Schlüsselwort)

angewendet und dient der Verwaltung von Abhängigkeiten, die zwischen Ergebnissen und bestimmten Annahmen bestehen.

Das KEEworlds-Modul ermöglicht den Einsatz **multipler Welten**, z.B. zum Verfolgen mehrerer alternativer Lösungswege mit der Möglichkeit des Backtracking. Die hierarchisch strukturierten Welten werden mittels des ATMS hinsichtlich ihrer Konsistenz überwacht.

Das Rulesystem-Modul umfaßt drei verschiedene Typen von **Regeln**, die alle als Objekte repräsentiert werden und hierarchisch organisiert werden können. Die Regeln werden in der an die englische Sprache angelehnten **TellAndAsk-Sprache** geschrieben. Sowohl Vorwärts- als auch Rückwärtsverkettung, jeweils mit verschiedenen Möglichkeiten der Agendasteuerung, werden zur Verfügung gestellt.

Das ActiveValues-Modul ermöglicht den Einsatz von Dämonen.

Das ActiveImages-Modul beinhaltet ein Paket von vordefinierten Grafiken, mit denen man u.a. die Werte von Objektattributen modifizieren oder darstellen kann. Diese Grafiken zeichnen sich durch besonders einfache Bedienbarkeit aus.

Das KEEpictures-Modul ermöglicht das Zeichnen von Bitmaps und anderen Grafiken, welche auch Hotspots[4] beinhalten können.

Mit Common Windows können über Lisp-Funktionen effiziente und portable Endbenutzer-oberflächen kreiert werden.

3.3. Wissensrepräsentationsformalismen

Wissensbasen

Als zentrales Konzept der Wissensrepräsentation dienen in KEE hierarchisch anzuordnende Objekte, welche auch zur Repräsentation von Regeln und Regelklassen verwendet werden. Alle Objekte werden Wissensbasen (*knowledge bases*) zugeordnet, wobei gleichzeitig – auch während einer Inferenz – mit mehreren Wissensbasen gearbeitet werden kann. Eine Wissensbasis kann auch Fakten beinhalten, die sich nicht auf Objekte beziehen (unstructured facts).

---

[4]das sind Bereiche der Graphik, deren Anklicken – wie ein Kommando – spezielle Reaktionen des Systems bewirken

Objekthierarchien

Objekte (***Units***) werden in Klassen und Instanzen unterteilt. Die einzigen zwischen Klassen untereinander und zwischen Klassen und Instanzen möglichen Relationen sind 'superclass/subclass' und 'member/member.of'. Dabei können Objekte auch mehrere übergeordnete Objekte (Eltern) haben. Die Objekthierarchie kann graphisch dargestellt werden.

Die Attribute der Objekte (***Slots***) können sowohl entlang der Relationen vererbt werden (*Member Slots*), als auch einzelnen Frames zugeordnet werden, ohne daß eine Vererbung der Attribute stattfindet (*Own Slots*). Für jede Relation stehen 12 verschiedene Vererbungsstrategien zur Spezifikation der Wertvererbung eines Slots zur Verfügung. Ein Selbstdefinieren von Vererbungsstrategien ist möglich, aber nicht dokumentiert. Slots können verschiedene Werte, u.a. auch Funktionen (Methoden) und Objekte, aufnehmen. Der Wertebereich des Slots kann, ebenso wie die Mindest- und Maximalanzahl der enthaltenen Werte, beschränkt werden. Für den Wertebereich stehen dem Benutzer neben allen in KEE verwendeten Datentypen (inklusive der Lisp-Datentypen) alle (selbstdefinierten) Frames, sowie 12 weitere Datentypen zur Verfügung. Alle Datentypen können beliebig miteinander kombiniert werden. Der Benutzer kann zwischen fünf verschiedenen Strategien wählen, wie das System auf Verletzungen der Werterestriktionen reagieren soll.

Der Wertebereich, die Mindest- und Maximalanzahl der Werte und die Vererbung eines Slots werden mit Hilfe von Attributen der Slots (***Facets***) festgelegt. Der Benutzer kann selbst beliebige Facets definieren.

Methoden (***Methods***) werden als Werte von Slots repräsentiert und grundsätzlich in Lisp programmiert. Es ist jedoch problemlos möglich, von Lisp aus die TellAndAsk-Sprache zu verwenden.

Dämonen (***Active Values***) werden als Units repräsentiert und können bei neun verschiedenen Slot-Operationen wie z.B. Lese-, Schreib-, Lösch-, Kopier-, Umbenennungsoperationen usw. getriggert werden. Mit Hilfe von Dämonen läßt sich in KEE u.a. die Funktionalität von Constraints realisieren.

Regeln

Regeln (***Rules***) und Regelklassen werden als Units repräsentiert, hierarchisch organisiert und damit zu Gruppen zusammengefaßt. Sie können sowohl vorwärts als auch rückwärts verkettet werden. Um die Effizienz größerer Regelsysteme zu steigern, können die Regeln von TellAndAsk in Lisp kompiliert werden.

In KEE gibt es drei verschiedene Regeltypen: Der erste ermöglicht Inferenzen, die die Wissensbasis unwiderrufbar modifizieren, der zweite ermöglicht Inferenzen unter Verwendung des ATMS (z.B. für nichtmonotone Logik) und der dritte ermöglicht Inferenzen unter Verwendung der multiplen Welten.

Die ***TellAndAsk***-Sprache läßt Abfragen und Assertionen betreffend Relationen, Slotwerte, Werte bestimmter Facetten und nicht in der Objekthierarchie verankerte Fakten (*unstructured facts*) zu. Zusätzlich kann jederzeit auf Lisp zurückgegriffen werden. Die Terme dieser Sprache können mit AND und OR verknüpft werden, sie können negiert und ineinander geschachtelt sowie mit einem Existenz- oder einem Allquantor versehen werden. Ferner ist es möglich, das System zu veranlassen, den Benutzer nach einem Faktum zu fragen, oder die weitere Inferenz durch Angabe einer Regelklasse zu steuern.

Reason Maintenance wird durch ein Assumption-based Truth-Maintenance-System (***ATMS***) realisiert, welches auf Werten von nicht vererbbaren Slots und auf den oben erwähnten unstructured facts arbeiten kann.

Die Verwendung von multiplen Welten (***worlds***) wird in KEE vom ATMS kontrolliert, ist dadurch aber nur sehr eingeschränkt möglich: Von Welt zu Welt können sich lediglich die Werte von nicht vererbbaren Slots und unstructured facts ändern. Durch eine graphisch darstellbare hierarchische Anordnung von Welten werden Fakten von Welt zu Welt vererbt. Welten können vereinigt (merged), ihre Vereinigung kann aber auch ausgeschlossen werden (exclusion sets).

Die Repräsentation von zeitlichem Wissen sowie das Arbeiten mit Wahrscheinlichkeiten wird nicht durch entsprechende Konstrukte unterstützt.

3.4. Inferenzmechanismen und Kontrollfunktionen

Regelinferenz

Als **Inferenzmechanismen** stehen sowohl Vorwärts- als auch Rückwärtsverkettung zur Verfügung. Jede Regel kann von beiden Mechanismen verarbeitet werden (in wenigen dokumentierten Spezialfällen werden allerdings einzelne Klauseln ignoriert) und muß daher nicht doppelt repräsentiert werden. Während eines Inferenzvorganges kann problemlos mehrfach zwischen den beiden Mechanismen und den für die Inferenz benutzten Regelklassen gewechselt werden. Der Benutzer kann während eines Inferenzvorganges vom System nach dem Zutreffen oder Nichtzutreffen von Klauseln gefragt werden.

Für die **Vorwärtsinferenz** stehen folgende Strategien zur Verfügung: breadth first, depth first, greatest premise complexity, greatest weight, least premise complexity, least weight, weighted greatest premise complexity, weighted least premise complexity. Zusätzlich kann der Benutzer selbstdefinierte Strategien verwenden. Für die **Rückwärtsinferenz** stehen folgende Strategien zur Verfügung: breadth first, depth first, best first (shortest goal stack). Auch hier kann der Benutzer zusätzlich selbstdefinierte Strategien verwenden. Ein graphischer Browser, Text-Trace und weitere Debugging-Möglichkeiten (Setzen von Breakpoints, Wiederherstellung des Ausgangszustandes ...) ermöglichen ein genaues Beobachten/Debuggen der Regelinferenz.

Kontrollfunktionen

Regelinferenzen werden von Lisp aus durch Aufruf der entsprechenden Funktion gestartet. Das bedeutet, daß sie auch von Methoden oder Dämonen und somit letztlich auch über ein graphisches Benutzerinterface gestartet werden können. Da die die Regelinferenz auslösenden Funktionen ein Ergebnis zurückliefern, können Regelinferenzen in beliebiger Reihenfolge in Lisp-Programme eingebunden und darüber koordiniert werden.

3.5. Entwicklungsunterstützung

Desktop

Die **Benutzeroberfläche** von KEE zeichnet sich durch ihre Komplexität und umständliche Bedienbarkeit aus. Sie läuft unter der graphischen Benutzeroberfläche X-Windows und besteht aus einem oder mehreren großen Fenstern, welche verschiedene kleinere Fenster mit unterschiedlichen Funktionalitäten zu einem sogenannten Desktop zusammenfassen. Es gibt u.a. Lisp Listener (Lisp Interpreter), Fenster für die Benutzerkommunikation (Typescript Window), ein Fenster für Kurzinformationen (Prompt Window), sowie Fenster für die Darstellung verschiedener Konstrukte wie z.B. Wissensbasen, Units, Slots usw. Der Benutzer kann den Desktop mit den zur Verfügung gestellten Fenstertypen, sowie mit KEEpictures und ActiveImages frei konfigurieren, abspeichern und jederzeit wieder laden. Standardfarben und -fonts für Desktops können vom Benutzer geändert und separat abgespeichert bzw. geladen werden (User Profile). Die intensive Menüverwendung bei Fensteroperationen, die nicht durch Short-Cuts abgekürzt werden kann, wirkt sich negativ aus (z.B. Verschieben eines Fensters auf dem Desktop). Der Benutzer ist auf die Desktops angewiesen, da die KEE-Wissensbasen zwar in einer ASCII-Datei abgelegt werden, dort aber nur mit großem Aufwand – wenn überhaupt – konsistent geändert werden kann.

Editieren

Namen von Wissensbasen, Objekten und Slots sind diese über Maus-Klick aktivierbar. Dabei erscheint ein Menü mit den entsprechenden Operationen. Es gibt jedoch für die einzelnen Konstrukte, wie z.B. Objekte, Regeln usw. keine eigenen **Editoren**. Möchte man Änderungen vornehmen, so wird ein Texteditor wie z.B. emacs mit dem zu editierenden Konstrukt geladen. Die darin getätigten Änderungen werden nach Beendigung der Editier-Session in die Wissensbasis übernommen. Die strikte Trennung zwischen Darstellungs- und Eingabefenstern ist hinderlich: läßt man sich z.B. einen Slot anzeigen, kann man die angezeigten Werte nicht direkt editieren, sondern muß über den Umweg eines Menüs und eines Eingabefensters (Typescript Window) gehen. Über ActiveImages – das sind Fenster, mit deren Hilfe man den augenblicklichen Zustand eines Slots wiedergeben und verändern kann – kann hier für einige häufig benutzte Slots Abhilfe geschaffen werden.

Lisp

Über eine Tastenkombination läßt sich die Parameterliste einer definierten Lisp-Funktion (auch einer selbstdefinierten) anzeigen. Alle Operationen, die der Benutzer über Menüs aufrufen kann, stehen auch über Funktionsaufrufe zur Verfügung und sind entsprechend dokumentiert. Das gleiche gilt für eine Vielzahl weiterer KEE-spezifischer Funktionen. Funktionsfiles werden außerhalb von KEE mit beliebigen Texteditoren erstellt. Als Debugger steht ein typischer Lisp-Debugger zur Verfügung.

Versionsverwaltung

Eine über die Protokollierung von Änderungsdaten hinausgehende Versionsverwaltung von Wissensbasen gibt es nicht. Erstellungs- und Änderungsdaten von Objekten und Wissensbasis werden automatisch direkt in den Objekten bzw. im File der Wissensbasis eingetragen.

Konfigurierbarkeit

In der KEE-Entwicklungsversion ist eine Konfigurierbarkeit, d.h. das Weglassen einzelner nicht benötigter Module, nicht vorgesehen. Dies wird erst durch ein anderes Produkt der KEE-Produktfamilie (Configurable KEE) möglich.

Grafik

KEE bietet eine Vielzahl graphischer Möglichkeiten. Mit **ActiveImages** und **KEEpictures** kann eine Oberfläche im 'rapid prototyping'-Verfahren erstellt werden. Für den späteren Einsatz empfiehlt selbst der Hersteller eine Reimplementierung unter Verwendung von Common Windows, einem Window-System als Erweiterung von Common Lisp. Der Benutzer hat über eine Vielzahl von Farben und graphischen Konstrukten inklusive Bitmaps, Hotspots, bewegter Grafiken, Grafiken mit Methodenfunktionalität, Grafiken zum Darstellen und Modifizieren von Slotwerten und selbstdefinierten Menüs überdurchschnittliche Möglichkeiten.

Schnittstellen

Eine Schnittstelle zu Datenbanken kann als Zusatzmodul gekauft werden, ebenso eine Schnittstelle zur Programmiersprache C. Schnittstellen zu Kappa und ProKappa sind geplant.

3.6. Dokumentation und Support

Support

Der Support wurde vom Autor niemals in Anspruch genommen. Nach Aussagen anderer Kunden ist dieser als kompetent und freundlich zu beurteilen. Neben dem Telefonsupport gibt es einen Support über e-mail. Es finden regelmäßige (jährliche) Users Group Meetings statt, die einen Austausch zwischen Anwendern und Mitarbeitern von IntelliCorp (auch der Entwicklungsabteilung) ermöglichen. Nach Aussagen des Herstellers ist allerdings die Existenz eines Supports für KEE nur noch für höchstens fünf Jahre gewährleistet. Der Entwicklungsschwerpunkt von IntelliCorp wurde auf die Werkzeuge Kappa und ProKappa verlegt, so daß – abgesehen von KEE 4.1 – keine weiteren Versionen von KEE zu erwarten sind.

Handbücher

Die Handbücher sind extrem umfangreich, aber gut strukturiert und mit einem guten globalen Index versehen. Es gibt ein Handbuch, welches den Benutzer an die Oberfläche heranführt, ein weiteres, welches eine grobe Zusammenfassung aller KEE-Module beinhaltet und die modulspezifischen Handbücher, die jeweils ein Modul erschöpfend behandeln. Ergänzend zu den Handbüchern ist ein KEEtutor erhältlich, der anhand eines umfassenderen Beispiels (Kriminalgeschichte) durch die gebräuchlichsten Konstrukte führt und einen wesentlichen Beitrag zum autodidaktischen Einarbeiten in KEE leistet.

3.7. Zusammenfassende Bewertung

KEE verfügt über eine – von der Beschränkung bei den Relationen einmal abgesehen – mächtige Objektrepräsentation und sehr gute Möglichkeiten zum objektorientierten Programmieren. Es verfügt über ein sehr gutes Regelsystem und hat exzellente Grafikmöglichkeiten. KEE ist ein sehr offenes System, da ein Teil von KEE selbst in vom Benutzer editierbaren Systemwissensbasen zur Modifikation und Ergänzung bereitgestellt wird.

KEE ist unter Modularisierungsaspekten aufgrund des problemlosen parallelen Handhabens mehrerer Wissensbasen, der Regelhierachisierung und der Trennung von Regeln und Inferenzsteuerung für das Projekt gut geeignet. Die Eignung für das Projekt wird dadurch stark reduziert, daß ein Weltenkonzept nur eingeschränkt und ein Zeitkalkül gar nicht vorhanden ist. Weitere große Nachteile sind die umständliche und nicht zu umgehende Benutzeroberfläche (Desktops), mit der unter der von uns verwendeten Windows-Version X11 Release 5 Laufunsicherheiten bestehen, und die fehlende Konfigurierbarkeit der getesteten Version, was der KEE-Entwicklungsversion eine gewisse Schwerfälligkeit verleiht.

4. Evaluation von Knowledge Craft

Knowledge Craft ist ein hybrides Lisp-basiertes Entwicklungswerkzeug zur Realisierung offener (erweiterbarer) wissensbasierter Systeme. Die Kernsprache CRL (Carnegie Representation Language) bildet die Basis für Komponenten, welche die Applikation verschieden spezifischer Wissensrepräsentationsformen, Problemlösungsstrategien, einer graphischen Benutzerschnittstelle sowie den Zugriff auf Datenbanken unterstützen. Der modulare Aufbau der Toolkomponenten erlaubt eine entwicklungsspezifische Konfiguration von Knowledge Craft. Für den Einsatz im Projekt wurde nur eine Version von Knowledge Craft getestet, die um die Komponenten CRL-Prolog, Timepak, Simpak und Statpak reduziert war. Für diese Komponenten konnte jedoch teilweise auf Erfahrungen aus früheren Projekten zurückgegriffen werden.

4.1. Produktbeschreibung

Die Entwicklung von Knowledge Craft wurde von der Carnegie Group, Inc. (USA) zuerst für Lisp-Maschinen - Symbolics und TI-Explorer - vorgenommen. Seit einigen Jahren ist Knowledge Craft auch auf Hardware-Plattformen wie DECstation, VAX/VMS-Rechner sowie SUN-3 und SUN Sparc Station einsetzbar. Während die Carnegie Group die Maintenance des Produktes übernimmt, obliegt die Zuständigkeit für den Vertrieb und Support in Deutschland der Danet GmbH. Als Softwarebasis werden Lucid Common Lisp oder Allegro Common Lisp vorausgesetzt.

Gemäß der Philosophie des Herstellers, daß das Design eines wissensbasierten Systems auf der Basis eines konzeptuellen Modells erfolgt, basiert der Aufbau des Werkzeuges auf der Kernsprache CRL. CRL legt als Grundelement das Konzept des *schema*, das der Frame-Repräsentation entspricht, zugrunde. Die Mächtigkeit der Wissensrepräsentationssprache ist einerseits durch die Integration der universellen Möglichkeiten von Lisp begründet, andererseits durch die Implementation aller Repräsentationsformalismen in CRL selbst. Vorausschauend sei hier auf die Repräsentationsmöglichkeiten mit Hilfe eines objektorientierten Programmierstils, multipler Welten - hier Contexts genannt -, Behaviors sowie eines umfassenden benutzer-definierbaren Relationensystems verwiesen.

Zur Wissensverarbeitung stellt Knowledge Craft die klassischen Sprachen Prolog und OPS5 in einer für CRL modifizierten Version bereit. Desweiteren stehen Moduln für die Prozeßsimulation (Simpak) und statistische Auswertungen (Statpak) zur Verfügung.

Gleichfalls kann auf eine taxonomisch spezifizierbare objektorientierte Programmierung zugegriffen werden.
Das Produkt unterstützt darüber hinaus Zeitrepräsentationsmöglichkeiten, eine SQL-Datenbankschnittstelle sowie die Entwicklung einer graphischen Benutzeroberfläche.

4.2. Produktarchitektur

Der modulare Aufbau des Tools wird bereits durch die nachfolgend abgebildete Systemarchitektur erkennbar:

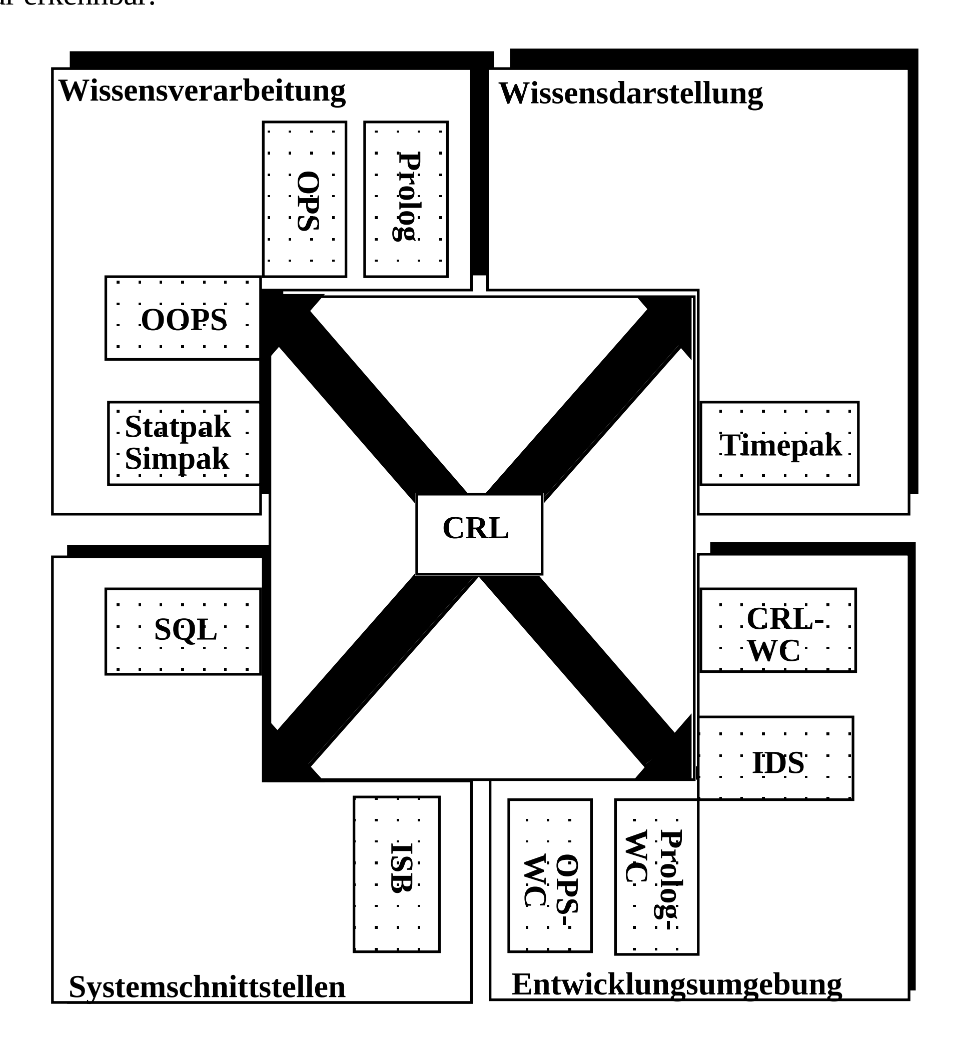


Abb. 3: Die Systemarchitektur von Knowledge Craft. (nach: *Knowledge Craft Informationsbroschüre,* Danet GmbH.)

Die **Wissensrepräsentationssprache CRL** stellt die Basiskomponente des Werkzeugs dar und integriert Moduln anhand nachfolgender funktionaler Gesichtspunkte:

Moduln zur **Wissensverarbeitung**:

CRL-OPS (eine modifizierte Version des Produktionensystems OPS5)
CRL-Prolog (eine modifizierte Version der logikbasierten Programmiersprache Prolog)
OOPS (ObjektOrientierter ProgrammierStil)
Statpak (wissensbasierte Verarbeitung von Statistiken)
Simpak (Prozeßsimulation)

Modul zur **Wissensdarstellung**:

Timepak (Repräsentationformen für Zeitdauer, -intervalle und -punkte)

Moduln, die der **Entwicklungsumgebung** zuzuordnen sind:

IDS (Interface Development System zur Generierung einer Benutzeroberfläche)
CRL-Workcenter (CRL-spezifische Arbeitsumgebung)
OPS-Workcenter (OPS-spezifische Arbeitsumgebung)
Prolog-Workcenter (Prolog-spezifische Arbeitsumgebung)

Moduln zur Definition einer **Systemschnittstelle**:

SQL (Schnittstellenmodul zu SQL (Structured Query Language-Datenbanken)
ISB (Integrated Schema Base zur Verwaltung sehr großer Wissensbasen, z.Zt. noch nicht auf SUN-Rechnern verfügbar)

Knowledge Craft besitzt einen offenen Aufbau. Zum einen kann der vollständige Funktionsumfang von Common Lisp direkt genutzt werden, zum anderen ist ein Einblick in das systemeigene, jedoch geschützte, konzeptuelle Modell der toolspezifischen Konstrukte möglich. Dieses steht immer zur Verfügung und wird um die zu modellierende Anwendung erweitert. Des weiteren besteht eine Abhängigkeit der jeweiligen Moduln des Werkzeuges (s.o.) nur gegenüber CRL, so daß aus technischer Sicht keine Interdependenzen oder Installationszwänge zwischen einzelnen Moduln bestehen. Daraus resultiert, daß ein customizing, d.h. eine den Zielvorgaben entsprechende, entwicklungsspezifische Konfiguration des Werkzeuges auf der Basis von CRL ohne weiteres möglich ist.

4.3. Wissensrepräsentationsformalismen

Das zentrale Konzept der Wissensrepräsentationssprache **CRL** ist das *schema*. Dieser framekonformen Darstellungsform liegen alle werkzeugeigenen strukturellen Konstrukte

zugrunde. Schemata können als Typ definiert oder physikalisch generiert werden. Gleichfalls existiert die deklarative Möglichkeit zur Unterbindung dynamisch erzeugbarer Interferenzen zwischen Schemata mit identischem Bezeichner.

Schemata werden durch *slots* attributiert. Der Wert *(value)* eines Slots kann faktische, relationale, funktionale Information oder eine Liste von Elementen beliebigen Typs beinhalten. Slots können einerseits wertespezifische Restriktionen auferlegt werden, andererseits kann der Zugriff auf einen slot - abhängig von dem ihm zugewiesenen Wert - mit einem freidefinierbaren Verhalten *(demons)* verknüpft werden. Diese sogenannten "Dämonen", die active values entsprechen, sollten als transparente Software-Monitore eingesetzt werden, da sie in Aktion nicht zu unterbrechen sind.

Ein wesentlicher Bestandteil von CRL ist das **Relationensystem**. Relationen (*relations*) werden als Schemata verstanden und sind aufgrund ihrer offenen Attributierbarkeit verschieden spezifisch definierbar. Es wird unterschieden zwischen systemeigenen und entwicklerdefinier–baren Relationen. Zu den systemeigenen Relationen zählen primär die *is-a* und *instance* Relationen, die Schemata als Klassen und deren Instanzen in einer Vererbungstaxonomie anordnen. Zur weiteren Strukturierung von Schemata stehen die Relationen *member-of, component-of* und *part-of* zur Verfügung. Zur Definition spezifischer Relationen stellt CRL dem Entwickler Formalismen zur Spezifikation von Map-Funktionen, eigenen Vererbungs–mechanismen sowie eine Path-Grammatik bereit.

CRL bietet gleichfalls eine Unterstützung zur Dokumentation und autorenspezifischen **Versionsverwaltung** der generierten Repräsentationskonstrukte. Das Hilfsmittel zur getrennten Darstellung von Information über die anwendungseigenen Wissenskonstrukte wird als *meta-knowledge* bezeichnet, welches zu diesem Zweck *meta-schemata, meta-slots* und *meta-values* bereitstellt.

Zur Darstellung **multipler Welten** werden innerhalb CRL Kontexte (*contexts*) bereitgestellt, die folgende Funktionen umsetzen:

- Simulation unterschiedlicher Hypothesen,
- Versionsverwaltung unterschiedlicher Wissensbasen.

Kontexte stellen virtuelle Kopien von Teil-Wissensbasen dar. Kontexte werden baumartig hierarchisiert, so daß ein untergeordneter Kontext (*child-context*) maximal alle Schemata seines direkt übergeordneten Kontextes (*ancestor-context*) erben kann. Schemata sind in einem child-context nur dann explizit zu definieren, wenn sie nur hier benötigt werden. Alle Manipulations-

möglichkeiten von Schemata stehen ebenfalls in Kontexten zur Verfügung. Zur Vermeidung von Inkonsistenzen sollten Manipulationen nur in Kontexten, die keine untergeordneten Kontexte besitzen, vorgenommen werden. Kontexte können erzeugt, kopiert, zusammengesetzt, überlagert und gelöscht werden.

In CRL ist ebenfalls die objektorientierte Programmierung integriert. Schemata sind zu diesem Zweck als Objekte (*objects*) und Slots als **Methoden** (*methods*) zu vereinbaren. Die Methode beinhaltet eine Prozedur, die z.B. Regelinterpreter, Ausgabefunktionen oder beliebige Lisp-Funktionen aufruft. Das objektorientierte Prinzip wird realisiert, indem eine Nachricht (*call-method*) an das Objekt die Prozedur der angesprochenen Methode aktiviert. Dies erlaubt das Referenzieren einer beliebigen Standardfunktion unabhängig vom Namen und der Implementierung dieser Funktion in einem bestimmten Objekt oder einer Klasse von Objekten. Kombiniert mit einem Vererbungsmechanismus ist es möglich, Prozeduren auf einem adäquaten Abstraktionsniveau zu definieren.

Die Darstellung von zeitbezogenem Wissen wird durch das Modul **Timepak** unterstützt. Mit Timepak ist es möglich Ereignisse in Bezug zu qualitativen und quantitativen Zeitpunkten und -intervallen zu setzen. Der bereitgestellte Zeitkalkül lehnt sich an den von Allen an und beinhaltet 13 mögliche Zeitrelationen. Der konzeptuellen Repräsentation zeitlichen Wissens in Timepak ist das Schema *time-object*, zugrundegelegt. Absolute Zeitrepräsentationen werden durch die Unterklassen *time-units*, *time-points*, *fixed-intervals* und *durations* an *time-object* gebunden. Dagegen wird die relative Zeitrepräsentation zunächst an das Schema time-event gebunden, welches eine Unterklasse der absoluten Zeitobjekte ist. Das Clustern von time-events ist möglich. Damit können Gewichtungen von Ereignissen innerhalb eines Zeitintervalls vorgenommen werden. Es ist relativ aufwendig, mit dem Allen'schen Zeitkalkül die genauen Zeitrelationen zwischen zwei Intervallen zu bestimmen, da eventuell lange Ketten von Referenzereignissen auftreten, die sich zusätzlich noch verzweigen können. Um mit dem vorhandenen Kalkül effizient umgehen zu können, müßte man entsprechend dem zu bearbeitenden Wissensgebiet die Zeitrelationen auf ein vernünftiges Maß reduzieren, ohne daß es hierbei zu einem größeren Verlust an sinnvollen und erforderlichen Schlußfolgerungen kommt.

4.4. Inferenzmechanismen und Kontrollfunktionen

Zur integrierten Wissensverarbeitung stehen die regelorientierten Moduln CRL-OPS und CRL-Prolog sowie die Moduln Simpak und Statpak zur Verfügung.

CRL-OPS ist eine erweiterte Version der regel- bzw. produktionenorientierten Programmiersprache OPS5 und zeichnet sich durch einen datengetriebenen vorwärtsverkettenden Inferenzmechanismus aus. Produktionen (Regeln) bestehen aus einem Bedingungsteil LHS (Left Hand Side) und einem Aktionsteil RHS (Right Hand Side). Bedingungselemente der LHS können als Schablonen verstanden werden, die mit Datenelementen des Working Memory's von CRL-OPS verglichen - gematcht - werden. Elemente des Working Memory können beliebige CRL-Schemata, also Objekte bzw. Klassen und Instanzen sein. Sollen sehr komplexe Funktionen ausgeführt werden oder soll auf Teilen der Wissensbasis gearbeitet werden, die nicht im Working Memory stehen, können Lisp-Funktionen in der LHS verwendet werden. Jedoch ist darauf zu achten, daß diese Ausdrücke Prädikatenfunktionen sind, die - aus Konsistenzgründen - den Zustand der Wissensbasis nicht ändern dürfen.

Auf der RHS werden Aktionen beschrieben, die den Inhalt des Working Memory ändern können. Entsprechende Aktionen können Eingabe- bzw. Ausgabeoperationen sein oder aus dem funktionalen Umfang von CRL entstammen. Auf der RHS können auch beliebige Lisp-Funktionen aufgerufen werden.

Die Entscheidung, welche Regel abgearbeitet wird (feuert), wird durch die Inferenzmaschine, die den Recognize-Act-Cycle realisiert, getroffen. Jeder Zyklus matcht die Bedingungselemente der LHS gegen die Datenelemente des Working Memory. Sind alle Bedingungselemente einer Regel erfüllt, wird die Regel in die sogenannte Konfliktmenge (conflict set) kopiert. Diese enthält sämtliche Regeln, deren Bedingungselemente alle vollständig erfüllt sind. Die Konfliktlösungsstrategie filtert die zu feuernde Regel gemäß des **RETE**-Algorithmus aus. Die von CRL-OPS zur Verfügung gestellten Konfliktlösungsstrategien sind MEA – in Anlehnung an das Problemlösungsverfahren "Means Ends Analysis" – und LEX (lexikalische Konfliktlösung). Darüber hinaus stellt CRL-OPS eine Reihe von Funktionen bereit mit Hilfe derer der Entwickler seine eigenen Konfliktlösungsstrategien definieren kann. Der Recognize-Act-Cycle endet, sobald keine Regel mehr in der Konfliktmenge vorhanden ist.

CRL-Prolog kombiniert die logikbasierte Regelsprache Prolog (Programming in Logic) mit der objektorientierten Wissensrepräsentationssprache CRL. Die Sprache Prolog arbeitet auf einem Teil der Prädikatenlogik erster Ordnung, basierend auf Horn-Klauseln. Eine Horn-Klausel (Prädikat) ist eine Implikation, die nur eine atomare Formel als Konklusion und in ihrer Prämisse mehrere durch UND verknüpfte Formeln enthalten darf. In Prolog werden Regeln abgearbeitet, indem die Prädikate eines Regelkörpers mit Axiomen und Regelköpfen nacheinander unifiziert werden. Die prinzipielle Regelabarbeitung erfolgt nach dem Resolutionsverfahren; damit ist es möglich, ein rückwärtsverkettendes Produktionensystem zu

erzeugen. Durch die Einbettung von Prolog in CRL steht der gesamte Funktionsumfang von CRL in CRL-Prolog zur Verfügung. Dabei ist von Interesse, daß CRL-Prolog-Programme als Schemata definiert und instanziiert werden und umgekehrt, daß Schemata in CRL-Prolog verarbeitet werden.

In Knowledge Craft können sowohl für CRL-OPS als auch für CRL-Prolog relativ effiziente Laufzeitsysteme kompiliert werden.

Die Moduln Simpak und Statpak wurden im Rahmen dieser Evaluation nicht getestet.

4.5. Entwicklungsunterstützung

Die Entwicklungsumgebung von Knowledge Craft beinhaltet einerseits Arbeitsumgebungen (*workcenter*) für CRL, CRL-OPS und CRL-Prolog, andererseits eine Umgebung zur Erstellung einer interaktiven Benutzeroberfläche (*IDS, Interface Development System*).

Beim **CRL-Workcenter** werden Schema-Strukturen mit dem *Palm Network Editor* in Baumform graphisch dargestellt. Da alle benutzerspezifischen Relationen verfügbar sind, kann ein semantisches Netz unter verschiedenen Aspekten visualisiert werden. Diese Arbeitsumgebung stellt auch einen Schema-Editor bereit.

Die Arbeitsumgebungen **CRL-OPS-** und **CRL-Prolog-Workcenter** bieten entsprechend den Erfordernissen der jeweiligen Sprache Editoren und Debugging-Möglichkeiten. Jedoch erweisen sich die angebotenen Arbeitsumgebungen für die Erstellung großer Wissensbasen als zu langsam und sind bestenfalls als ein Hilfsmittel zur Veranschaulichung von Wissensbasisausschnitten geeignet. Dem erfahrenen Entwickler sei zur Implementierung der Einsatz eines Lisp-Editors empfohlen.

Das **Interface Development System** ist gegliedert in das *Command-System* zur Entwicklung von I/O-intensiven Benutzer-Schnittstellen und in das *Window/Graphics-System* zur Applikation einer hardwareunabhängigen graphischen Benutzeroberfläche auf der Basis der Fenstertechnik. Dazu stehen dem Entwickler graphische Primitive und verschiedene Funktionen zur Manipulation von Items (bspw. Skalierung, Transformation und Zooming) zur Verfügung. Eine **SQL-Schnittstelle** zur Anbindung von relationalen Datenbanken ist als zusätzliches Modul vorhanden; ebenso existiert standardmäßig eine Schnittstelle über Common Lisp zu C. Die Einbindung von C-Funktionen kann nur für sehr zeitintensive Berechnungen (z. B. komplexe Matrizenmultiplikationen) empfohlen werden.

4.6. Dokumentation und Support

Der telefonische Support kann in den meisten Fällen als kompetent und freundlich beurteilt werden. Daneben gibt es die Möglichkeit, Informationen per e-mail sowie schriftlich einzuholen. Es wird darauf verwiesen, in Notfällen die Hotline der Kooperationsfirma in Deutschland zu kontaktieren und nicht die Herstellerfirma in den USA. In regelmäßigen Abständen werden Informationsbroschüren zu Verbesserungen und Neuentwicklungen zugesandt. Bezogen auf den Informationsgehalt sind diese jedoch für den interessierten Entwickler sehr knapp bemessen.

Ein neues Release von Knowledge Craft ist nach Aussage des Vertriebs im nächsten dreiviertel Jahr nicht zu erwarten.

Die Handbücher sind sehr umfangreich, aber nur mit Hilfe des Documentation Guide ist der Inhalt der verschiedenen Handbücher für gezielte Fragestellungen zugänglich. Das Updating der Handbücher erfolgt selektiv und erfordert eine gewisse Erfahrung des strukturgerechten Einordnens.

4.7. Zusammenfassende Bewertung

Knowledge Craft verfügt über eine sehr mächtige Wissensrepräsentationssprache, in welcher insbesondere das flexible Relationensystem, der Vergleich von Hypothesen mit Hilfe von verschiedenen Kontexten und die sehr guten Möglichkeiten der objektorientierten Programmierung überzeugten. Obwohl die Repräsentation und Propagierung von Constraints nicht explizit vorhanden ist, können elementare Constraints auf der Basis von Slotrestriktionen nachgebildet und durch Dämonen initialisiert und propagiert werden.

Regeln in CRL-OPS und CRL-Prolog werden nicht als Schemata repräsentiert. Die Integration der klassischen regelbasierten Sprachen OPS5 und Prolog in CRL erweist sich jedoch als vielseitig in der Wissensverarbeitung mit Hilfe unterschiedlicher Problemlösungsstrategien. Es ist davon auszugehen, daß die Effizienz einer Anwendung bei zunehmendem Umfang und steigender Komplexität von zu unifizierenden Prologregeln spürbar abnimmt.

Der modulare Aufbau des Tools ermöglicht eine individuelle Konfiguration von Komponenten für verschiedene Problemlösungsstrategien und zur Entwicklung von Benutzerschnittstellen

über der gemeinsamen Repräsentationskomponente, welche die Basissprache Common Lisp vollständig integriert.

5. Evaluation von ProKappa

ProKappa ist ein C-basiertes Werkzeug mit graphischer Entwickleroberfläche zur Erstellung wissensbasierter Systeme. ProKappa unterstützt die objektorientierte Programmierung, das regelbasierte Schließen und den Zugriff zu SQL-Datenbanken.

5.1. Produktbeschreibung

ProKappa ist speziell für die Erstellung von Anwender-Software unter UNIX konzipiert worden. ProKappa stellt mit der Sprachkomponente ProKappa C eine Erweiterung des Sprachstandards ANSI C dar und verfügt über den vollen Funktionsumfang zur objektorientierten Programmierung. Mit ProTalk beinhaltet ProKappa eine eigene Wissensrepräsentationssprache, die ein umfangreiches Regelsystem bereitstellt. Alle Erweiterungen, die ProKappa bietet, werden in ANSI C übersetzt.

ProKappa und Kappa-PC sind Produkte der Firma IntelliCorp und sollen im Jahr 1993 zu einem gemeinsamen Produkt Kappa zusammengefaßt werden. Derzeit besteht bereits eine Testversion, welche die Kommunikation zwischen dem UNIX-basierten ProKappa und dem unter Microsoft-Windows lauffähigen Kappa-PC realisiert. Dadurch wird die Entwicklung eines auf mehrere Rechner verteilten Expertensystems ermöglicht, die über Nachrichten miteinander kommunizieren können. Die Maintenance des künftigen Produkts Kappa obliegt der Firma IntelliCorp.

5.2. Produktarchitektur

ProKappa besteht aus einer **interaktiven Entwicklerschnittstelle** zu der ProKappa C Workbench. Hierbei handelt es sich eine interaktive C-Umgebung mit integriertem C-Compiler und Linker. Dadurch ist es möglich C-Code und ProTalk-Code im Interpreter-Modus zu laden, zu debuggen und auszuführen. Der C-Code wird beim Laden in die ProKappa C Workbench zunächst nur interpretiert, wohingegen ProTalk-Code beim Laden in ANSI C übersetzt und anschließend als C-Code interpretiert wird. ProKappa erweitert. ANSI C um einen **Objektmanager** und einen **Regelmanager** und um die für die objektorientierte Programmierung nötigen Datentypen. Des weiteren sind Tools zur Entwicklung eines graphischen Endbenutzerinterfaces vorhanden. Es steht ein Compiler zur Verfügung, der

ProKappa C und ProTalk in ANSI C übersetzt, das dann mit einem beliebigen ANSI C-Compiler zu einer lauffähigen Version kompiliert werden kann.

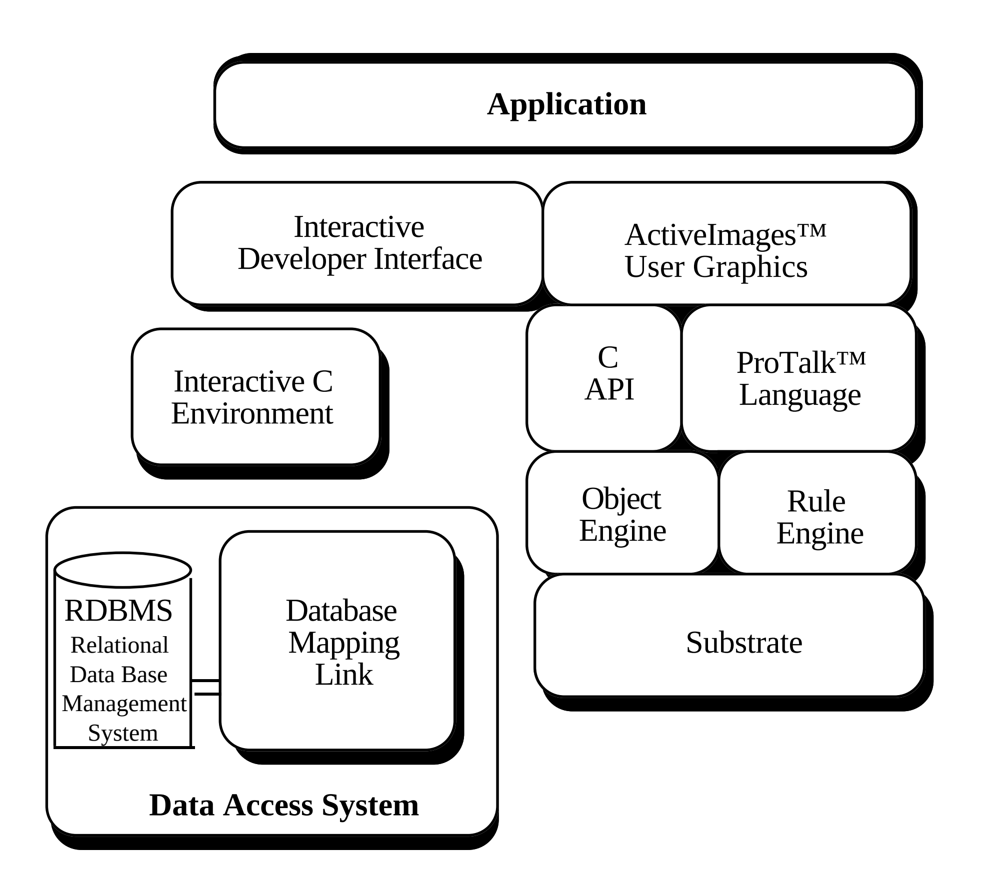


Abb. 4: Die Systemarchitektur von ProKappa (nach: *ProKappa User's Guide*, IntelliCorp Inc.)

In ProKappa sind nicht alle Konstrukte sind als Objekte (Frames) definierbar und entsprechend manipulierbar. Regeln werden nicht als Objekte, sondern in der ProKappa spezifischen Sprache *ProTalk* repräsentiert; weitere Konstrukte wie ***Application***[5], ***Module*** usw. können als Objekte definiert werden.

5.3. Wissensrepräsentationsformalismen

ProKappa stellt Objekte als einen framebasierten Formalismus zur Repräsentation von Wissen bereit. Diese können Daten oder Behaviours beinhalten. Das Objekt System wird durch eine umfassende Bibliothek von Funktionen für das Erzeugen und Manipulieren von Objekten

[5]"Application" ist ein ProKappa–spezifischer Begriff, der verschiedene Module, Regelmengen und Source-Code-Files bezeichnet

unterstützt. Gleichfalls besteht die Möglichkeit, Objekte in einer hierarchischen Taxonomie, die Klassen, Subklassen und Instanzen definiert, zu repräsentieren. Objekthierarchien können dynamisch erzeugt und modifiziert werden.
Ein **Objektframe** kann mehrere Slots haben, die seine Eigenschaften in Form von Werten enthalten können. **Slots** können nur auf der Ebene der Objektklassen und nicht auf der Instanzen-Ebene erzeugt werden. Gleichfalls sollte im Vorfeld bekannt sein, welche Aufgabe ein Slot übernimmt. Vor der Generierung eines Slots muß eine Entscheidung für eine der drei Slotarten *Single-value-* (der Slot enthält einen Wert), *Multi-value-* (der Slot enthält mehr als einen Wert) oder *Method-slot* (der Slot enthält eine Prozedur) getroffen werden. Die Defaultvorgabe ist ein Single-value-Slot. Die Entscheidung für einen der o. g. Slotarten wird durch die Vererbungsrolle des entsprechenden Slots spezifiziert. Die Vererbungseigenschaft eines Slots sieht folgende Slottypen vor: *Default-Slots*, *Own-Slots* und *Subclass-Slots*.

Default-Slots werden an Klassen und Instanzen vererbt. Own-Slots hingegen sind lokal, d.h. sie werden nicht vererbt. Subclass-Slots werden nur an Unterklassen vererbt. Die Vererbung der Slots bzw. Slotwerte wird beim Erzeugen eines Slots bzw. bei der Wertzuweisung an einen Slot durch Vorwärtspropagierung erreicht.

Es gibt in ProKappa keine expliziten Defaults für Slotwerte. Die Slotwerte der übergeordneten Objekte stehen jedoch den Slots der Subklassen bzw. Instanzen zur Verfügung, falls sie den Wert "?" besitzen. Diese Slotwerte können durch lokale Werte überschrieben werden. Werden diese lokalen Werte wiederum gelöscht, stehen die Parent-Werte erneut zur Verfügung.

ProKappa unterstützt verschiedene Methoden zur **multiplen Vererbung**. Bei Slot-Vererbung erhält das untergeordnete Objekt alle Slots der übergeordneten Objekte. Dabei kommt natürlich jeder Slotname nur einmal vor, d.h. wenn zwei übergeordnete Objekte beide einen Slot 'Slot1' haben, hat das untergeordnete Objekt diesen Slot nur einmal. Bei der Wertvererbung gibt es im wesentlichen zwei Vererbungsmethoden: *Self-first-union* und *Self-last-union*. Bei der Self-first-union wird eine Vereinigungsliste aller geerbten und der lokalen Werte gebildet, wobei die lokalen Werte am Listenanfang stehen. Entsprechend wird bei der Self-last-union eine Vereinigungsliste aller geerbten und der lokalen Werte gebildet, wobei die lokalen Werte am Listenende stehen. Bei multipler Vererbung sind keine eigenen Vererbungsstrategien programmierbar.

Für Wertrestriktionen steht das *ActiveRelations System* zur Verfügung. Das Nachladen des ActiveRelations System muß entweder explizit vorgenommen oder in den 'initprokappa'-File integriert werden.

Facetten zur Slot- und Wertkontrolle können einerseits vom Entwickler definiert werden, andererseits legt das ActiveRelations System bestimmte Facetten automatisch an. Diese werden genauso definiert wie Slots und sind an den zu kontrollierenden Slot gebunden.
Das ActiveRelations System stellt zwei Hilfsmittel zur Spezifikation von Werten eines Slots zur Verfügung:

1. Value Type: erlaubt die deklarative Beschreibung eines Slots, dessen Wert ein ProType, d.h. ein ProKappa eigener Typ, oder eine Liste von Werten sein kann. Werteverletzungen werden wahlweise durch Warnmeldungen, Dialogbox, Unterdrückung oder durch selbst geschriebene Funktionen behandelt. Mit der Value Type Deklaration steht eine einfache Möglichkeit zur Beschränkung von Werten zur Verfügung.

2. Value Check: erlaubt die prozedurale Validierung von Slotwerten mit Hilfe von ProTalk Fragmenten. Es stellt eine flexibles Mittel zur Werteüberwachung dar (Monitoring). Das Erzeugen der Monitorprozedur und die Bearbeitung von Werteverletzung obliegt dem Entwickler.

Zur Beschreibung von Relationen zwischen zwei Objekten steht durch das **ActiveRelation System** das Konstrukt des *Slot Inverse* zur Verfügung. Es besteht die Möglichkeit der one-to-one, one-to-many, many-to-one und many-to-many Relation zwischen Objekten. Der Entwickler kann dabei selbst Relationen auf den Objekten definieren.

Ein weiteres Konstrukt - slot formula - erlaubt die Manipulation des Wertes eines zugehörigen Slots bei Zugriff auf diesen Slot. Ein slot formula besteht aus einer Folge von ProTalk Befehlen, die Slotwerte anderer Objekte zur Manipulation des zu berechnenden Slotwertes heranziehen können.

Für Slotoperationen können zudem vordefinierte **Monitore** *before change*, *after change* und *when-needed* eingesetzt werden.

5.4. Inferenzmechanismen und Kontrollfunktionen

Zur Wissensverarbeitung stehen einerseits Funktionen, Monitore und Methoden zur objektorientierten Programmierung, andererseits ein Inferenzmechanismus zur Vorwärts- und Rückwärtsverkettung von Regeln zur Verfügung. In ProKappa werden Regeln nur in der Sprache ProTalk geschrieben.

In **Regeln** wird der Zugriff auf alle Objektarten von ProKappa wie Objekte, Applications, Module, Slots, Slotwerte, Facetten und Facettenwerte ermöglicht. Weiterhin kann auf ausführbare Funktionen, die in C oder ProTalk geschrieben sind und auf alle weiteren bereitgestellten Funktionen, Operatoren und Relationen zugegriffen werden. Die genannten Formalismen sind sowohl im linken als auch im rechten Regelteil ausführbar. Objekte können sowohl auf der rechten als auch auf der linken Regelseite dynamisch erzeugt und gelöscht werden. Des weiteren kann eigener C-Source-Code in vollem Umfang in die Regeln integriert werden.

Im Regelantezedens gibt es die Möglichkeit, Klauseln disjunktiv oder konjunktiv zu verknüpfen. Über einen all-Operator kann eine Operation iterativ über alle Elemente einer Liste ausgeführt werden. Beispielsweise können alle Lösungen einer Goal-Anfrage der Art "all ?list = find Klausel(?X);" protokolliert werden.

Die Ordnung von Regeln in **Regelmengen** erlaubt eine hierarchische Regelstrukturierung. Bei Inferenzbeginn (forward oder backward) muß die zu inferierende Regelmenge spezifiziert werden. Es werden automatisch die Regelmengen mitbenutzt, die sich in der Hierarchie unter der spezifizierten befinden. Die zu inferierende Regelmenge kann durch C bzw. ProTalk berechnet werden. Dadurch kann leicht eine "Fokussteuerung" geschrieben werden, die je nach Bedarf verschiedene Teilmengen der gesamten Regelmenge erzeugt und der Inferenzmaschine zur Verfügung stellt.

Regeln können nicht gleichzeitig vorwärts als auch rückwärts verkettet werden. Um **Vorwärts-** und **Rückwärtsverkettung** auf eine Regel anwenden zu können, muß zunächst eine Kopie von ihr angelegt werden. Die Defaultstrategie bei der Rückwärtsverkettung ist depth first; sie erlaubt eine Prioritätsangabe der Regeln. Das Werkzeug unterstützt nicht die Programmierung von Strategien zur Vorwärts- oder Rückwärtsverkettung durch den Entwickler.

5.5. Entwicklungsunterstützung

In ProKappa werden Regeln, Funktionen und Methoden als Quelltext eingegeben und editiert.

Es gibt keinen speziellen Regeleditor; Regeln können nur als ProTalk-Code mit jedem beliebigen Editor bearbeitet werden. Bei der Verkettung von Regeln besteht die Möglichkeit, mit einem Debugger die Regeln Schritt für Schritt durchzugehen. Anschließend können die

einzelnen Schritte mit den jeweiligen Variablenbelegungen in beliebiger Reihenfolge und beliebig oft angesehen werden. Leider ist ein so erstellter Trace nicht speicherbar. Schleifen bei der Regelverarbeitung werden nicht automatisch erkannt.
Methoden werden als C- oder ProTalk-Code implementiert. Für die Parametereinstellung von Methoden bzw. zur Methoden-Aktivierung für einen Slot steht ein Editor zur Verfügung.

Der **graphische Frame-Editor** ist eine der Stärken von ProKappa. In diesem kann eine Objekthierarchie einfach bearbeitet werden. Des weiteren stehen alle wichtigen Informationen in verschiedener Ausführlichkeit zur Verfügung. Die Zwischenzustände beim Arbeiten mit der graphischen Oberfläche lassen sich bequem speichern und laden. Jedoch bietet ProKappa keine automatische Versionsverwaltung.

Das **Sichern/Laden** von Objekthierarchien gestaltet sich etwas unpraktisch. Es gibt die Möglichkeit, mit den Funktionen PrkSaveAsciiApp() und PrkLoadAsciiApp() eine Ascii-Version der Objekthierarchie zu sichern oder zu laden und sie im Editor zu bearbeiten. Dabei gehen allerdings wesentliche Informationen verloren. Beispielsweise werden Regelfiles beim Laden einer Applikation nicht mitgeladen. Es besteht aber die Möglichkeit, mit Hilfe der Systemfunktionen eigene Routinen zu schreiben und somit das standardmäßige Sichern und Laden mit Funktionen des Ascii-Save/Load zu kombinieren, ohne daß hierbei Inkonsistenzen auftreten.

Online-Help-Funktion sind nur rudimentär vorhanden, dagegen werden Fehlermeldungen und Warnungen ausführlich gegeben.

Zur Laufzeit können mehrere Wissensbasen sowohl vom Benutzer interaktiv als auch automatisch hinzugeladen werden. Die Anwendung einer Hierarchie von Wissensbasen kann realisiert werden, indem mehrere Applikationen geladen werden und jede Applikation in Module aufgeteilt wird. Mit Hilfe deklarativer Menühierarchien kann ein graphische Endbenutzer-schnittstelle erstellt werden.

5.6. Dokumentation und Support

Zur Qualität der Dokumentation sei bemerkt, daß ein Tutorial vorhanden ist, mit dessen Hilfe ein guter Überblick über die Möglichkeiten von ProKappa erreicht werden kann. Leider sind die Handbücher teilweise unvollständig; so fehlen beispielsweise im Inhaltsverzeichnis angegebene Teile.

Bezogen auf das Produkt ProKappa kann keine Aussage über den technischen Support der Firma IntelliCorp (München) getroffen werden. Erfahrungen mit dem Werkzeug Kappa-PC der selben Firma zeigen, daß Hilfestellungen gerne gegeben werden. Auftretende Probleme, die nicht direkt am Telefon geklärt werden können, werden per Fax beantwortet.

5.7. Zusammenfassende Bewertung

Die Qualität von Fehlermeldungen in ProKappa ist gut. Auch verhielt sich das Werkzeug während der Tests weitgehend stabil. Jedoch ist zu bemerken, daß das Rücksetzen während des Regeltraces zweimal an verschiedenen Stellen im Programm (im C-Listener und im Object-Browser) wiederholt werden mußte.

ProKappa kann als ein interessantes Werkzeug zur Erstellung wissensverarbeitender Systeme gesehen werden. So bieten die Editoren und Browser für die Objekthierarchie eine Vielzahl an Bearbeitungsmöglichkeiten. Gleichfalls stellt das ActiveRelations System angemessene Unterstützung zur Wissensrepräsentation bereitgestellt. In Hinblick auf das Projektvorhaben HYPERCON liegen die Schwächen von ProKappa in der fehlenden Möglichkeit, zum einen Strategien zur Regelverarbeitung frei zu definieren und zum anderen multiple Welten zu repräsentieren und zu verarbeiten.

6. Evaluation von ROCK

6.1. Produktbeschreibung

ROCK ist ein C++-basiertes Werkzeug, das den Anforderungen des IMKA-Standards (Initiative for Managing Knowledge Assets) an ein Wissensrepräsentationssystem entspricht. Zur Repräsentation von Wissen bietet ROCK eine framebasierte Komponente an, die über mächtige Konstrukte zur Definition von Objekttaxonomien, Relationen, multiplen Welten sowie zur Verwaltung von Metawissen wie etwa Versionskennung und zur Organisation von Wissensbasen verfügt. ROCK bietet die Möglichkeit der objektorientierten Programmierung, der Implementation von Slot Behaviours und von entwicklerdefinierten Vererbungsmechanismen. Jedoch stehen darüberhinaus keine weiteren Inferenzmechanismen zur Entwicklung eines wissensbasierten Systems bereit, insbesondere steht keine regelbasierte Komponente bzw. Problemlösungssprache zur Verfügung.

6.2. Produktarchitektur

In Abb. 5 wird die Architektur des Entwicklungswerkzeuges ROCK dargestellt:

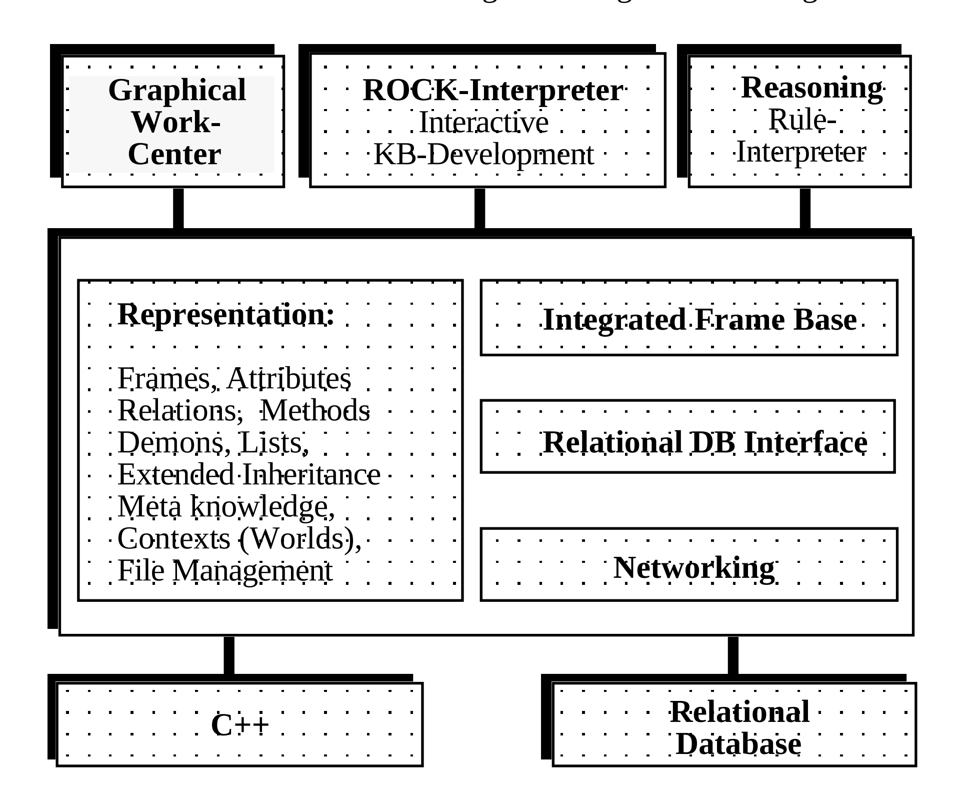


Abb. 5: Die Systemarchitektur von ROCK (nach: *Danet Informationsbroschüre*, Danet GmbH)

Für den Einsatz von ROCK ist ein **C++-Compiler** erforderlich, der jedoch nicht zum Lieferumfang des Werkzeugs gehört. Eine Wissensbasis kann sowohl durch Editieren und Laden einer Textdatei in einer speziellen Beschreibungssprache (frame definition language) erstellt als auch interaktiv oder aus einem Anwendungsprogramm heraus erstellt oder geändert werden. Dies geschieht mit Hilfe von speziellen C++-Funktionen zur Wissensrepräsentation, die ROCK zur Verfügung stellt. ROCK besitzt einen **Interpreter RI** (ROCK Interpreter), der keine eigene graphische Entwickleroberfläche bereitstellt. Um eine durch den Entwickler editierte Objekthierarchie anwenden zu können, muß der gesamte RI erneut compiliert und eingebunden werden, damit die entwicklerspezifische Objekthierarchie zur Verfügung steht.

ROCK bietet eine Schnittstelle zu relationalen Datenbanken an, um externes Faktenwissen in eine Applikation zu laden.

Darüberhinaus steht die **Integrated Frame Base** (IFB) zur Verfügung, die es ermöglicht, auf Teile einer großen Objekthierarchie, die beispielsweise als Datenbank angelegt ist, zuzugreifen. Mit Hilfe der IFB können mehrere Applikationen eine Objekthierarchie gleichzeitig und auf verschiedene Weisen benutzen. Dabei werden standardmäßig nur die für eine Applikation jeweils benötigten Teile in den Hauptspeicher geladen. Der Entwickler kann jedoch in den Vorgang des Ladens (und Sicherns) steuernd eingreifen, indem er bspw. geschlossen zu ladende/zu sichernde Teile (wie einen Teil der taxonomischen Hierarchie) angibt sowie eine Wiederauslagerung nicht mehr benötigter Teile vorschreibt.

Des weiteren existiert eine Schnittstelle, welche die Kommunikation zwischen verschiedenen ROCK-Applikationen im Rahmen einer Client-Server-Struktur ermöglicht (Networking).

6.3. Wissensrepräsentationsformalismen

ROCK bietet als grundlegende Wissensrepräsentationsformalismen objektorientierte Frame-hierarchien an, die dynamisch erzeugt, gelöscht und verändert werden können.

Frames werden in einer Taxonomie eingeordnet und besitzen Slots, die Werte beinhalten können. Dabei sind drei Slottypen zu unterscheiden: Attribute-, Relation- und Message-Slots. Relation-Slots sind für den Eintrag von Klassen- oder Instanzframes vorgesehen; bei diesen Frames wird normalerweise automatisch ein Rückverweis erzeugt. Über die von ROCK vordefinierten Relationen hinaus kann der Entwickler selbst weitere definieren und sehr detailliert die Vererbung von Eigenschaften vorgeben, die zwischen Frames stattfinden soll, die

in einer solchen Relation zueinander stehen. Ein Message-Slot enthält eine entwicklerdefinierte C++-Funktion, die für die Behandlung von Nachrichten verantwortlich ist, die dem durch das Frame definierten Objekt geschickt werden können. An weiteren Angaben zu Slots verlangt ROCK u.a. die Cardinality der Slotwerte (single/multiple) und ihren Typ; für Relationen kommt der Name der inversen Relation hinzu.

Die Werte von **Attribute-** und **Message-Slots** werden automatisch entlang der vordefinierten Relationen *subclassOf* bzw. *instanceOf* stets auf Unterklassen bzw. Instanzen vererbt. Die Vererbung kann jedoch durch Eintragen des ausgezeichneten Wertes "NULL_VALUE" in einen Slot unterbrochen werden. Der Vererbungsmechnismus wird in ROCK erst bei einer Anfrage an einen Slotwert aktiv (Query-time-inference). Dabei wird ein Algorithmus benutzt, der bei den Frames der übergeordneten Klassen (Parents) nach vererbbaren Slots bzw. Slotwerten sucht. Dabei kann man zwischen depth-first und breadth-first als Suchstrategie wählen.

Multiple Vererbung wird nicht unterstützt: zwar kann eine Klasse mehrere Oberklassen besitzen, doch wird bei der Suche nach einem Slotwert stets der erste gefundene Wert benutzt und die Suche sofort beendet.

ROCK bietet nicht die Möglichkeit, **Slots** explizit mit Defaultwerten zu belegen. Die normalerweise durch Vererbung in Unterklassen bzw. Instanzen übernommenen Slotwerte der Parents können jedoch durch Angabe lokaler Werte überschrieben werden. Werden diese lokalen Werte gelöscht, werden wieder die Werte der Parents benutzt. Es ist möglich, diese Werte zu lokalen Slotwerten von Unterklassen oder Instanzen zu machen ("Caching"). Für Slots müssen stets Wertetypen festgelegt werden. Dafür kommen ROCK- und C++-Typen in Frage, jedoch auch vom Benutzer definierte C++-Klassen ("UD_type").

Slots können (auch mehrere, nacheinander zu aktivierende) **Dämonen** (selbst programmierte C++-Funktionen) zugeordnet werden, die vor oder nach Lese-, Schreib- oder Löschzugriffen getriggert werden. Diese Zuordnung gilt im einfachsten Fall gleichmäßig für alle Vorkommen des Slots bei allen Frames, kann jedoch für Teile der Hierarchie spezialisiert werden.

Wertfacetten stehen nicht zur Verfügung. Die entsprechende Funktionalität läßt sich aber mit Hilfe von sog. **Meta-Knowledge** erreichen, das in einer unabhängigen Objekthierarchie repräsentiert wird. Die Frames dieser Meta-Knowledge-Hierarchie können Frames, Slots oder Slotwerten der eigentlichen Objekthierarchie zugeordnet werden. Meta-Knowledge kann dazu benutzt werden, Erweiterungen wie z.B. Repräsentation und Verwaltung von Unsicherheiten, Constraint Propagation oder Wertrestriktionen zu implementieren.

6.4. Inferenzmechanismen und Kontrollfunktionen

Der Entwickler kann selbst **Relationen** definieren und sehr genau steuern, welche Slots entlang der dadurch bewirkten Bezüge (Links) vererbt werden (alle, keine, alle einer expliziten Liste, alle außerhalb einer expliziten Liste, alle, für die eine selbstdefinierte Testfunktion einen bestimmten Wert liefert). Die Vorgabe ist zunächst möglich durch Erzeugen und Parametrieren eines Frames der Klasse *kb_Relation* mit dem Namen der Relation. Hier wird das Verhalten einheitlich für alle Vorkommen der Relation beschrieben. Das Verhalten kann jedoch für Relationen, die in Frames unterhalb einer Klasse vorkommen, geändert werden, indem dem Relationsslot dieser Klasse explizit ein weiteres Kontrollframe zugeordnet wird. Durch Aufruf vordefinierter Funktionen läßt sich der Ursprungsframe eines von einem untergeordneten Frames ererbten Werts bestimmen.
In ähnlicher Weise lassen sich die einem Slot zugeordneten Dämonen allgemein und für Teile der Framehierarchie speziell festlegen. Eine automatische Kumulation, bei der etwa zuerst die dem Slot im speziellen Frame zugewiesenen Dämonen und dann die dem Slot allgemein zugewiesenen Dämonen ausgeführt werden, ist nicht vorgesehen.

Zur genauen Beschreibung von Beziehungen zwischen Frames steht eine Grammatik zur Verfügung, mit der man die Abfolge von Relationen längs interessanter Pfade beschreiben kann. Dies erlaubt z.B. eine einfache Suche aller (auch indirekter) Teile eines Aggregats über einer Relation ist-Teil-von.

ROCK bietet die Möglichkeit, **verschiedene Welten** ("Kontexte") für die aktuelle Framehierarchie zu behandeln. Zu einem gegebenen Kontext kann man Nachfolgekontexte bilden; die Kontexte bilden bzgl. der Nachfolgebeziehung einen Baum. In einem Nachfolgekontext stehen sämtliche Klassen und Instanzen des Vorgängerkontexts unverändert zur Verfügung, soweit sie nicht explizit im Nachfolgekontext geändert oder gelöscht werden. Solche Änderungen und Löschungen sind nur in Blattkontexten möglich; sie läßt die Frames in allen andern Kontexten unverändert. Speicherökonomie wird dadurch erreicht, daß nur dann eine lokale Kopie eines Frames in einem Nachfolgekontext erzeugt wird, wenn es sich von dem entsprechenden Frame im Vorgängerkontext unterscheidet.
Sofern nicht Frames mit gleichem Namen gleichzeitig geladen werden, ist die Konsultation mehrerer Wissensbasen im Laufe einer Problemlösung möglich. Allerdings wird ohne die Verwendung der IFB Speicher nicht wieder freigegeben. Beim Laden eines Objekt-Definition-Files stürzt das System ab, wenn zu viele Fehler in dem File sind. Dieses Verhalten ist besonders kritisch, da im Verlauf einer Problemlösung mehrere Objekthierarchien nacheinander geladen werden können.

6.5. Entwicklungsunterstützung

Objekthierarchien lassen sich einfach erstellen, dabei können alle Parameter, Werte und Einstellungen im Zusammenhang mit der Definition einer Framehierarchie angegeben werden. Aber ein Zustand nach der Arbeit mit dem RI (z.B. mit veränderten Slotwerten oder noch wichtiger mit veränderter Objekthierarchie) kann zwar gespeichert und wieder geladen werden, das dabei erzeugte Format ist aber nicht editierbar. Die Möglichkeiten, die beim Editieren von Frame-Definition-Files zur Verfügung stehen, bietet im wesentlichen (wenngleich weniger komfortabel) auch der (interaktive oder programmgesteuerte) Aufruf von Funktionen bei geladener Wissensbasis. Das ist insbesondere für die Erzeugung von Instanzen gemäß aktuellen Aufgaben oder gemäß Einträgen in Datenbanken nützlich; der auch mögliche interaktive Eingriff z.B. in die Vererbungsvorschriften dürfte allerdings zu undurchschaubaren Systemzuständen führen. Beim Editieren kann der Entwickler insbesondere leichter über Namen von Frames Bezüge herstellen als beim Eingriff in eine geladene Wissensbasis.

Es gibt weder einen komfortablen **Frame-Editor** noch einen graphischen Browser für die Framehierarchie. Die einzige Möglichkeit, die Framehierarchie sichtbar zu machen, besteht in einem rudimentären Tree-Trace, dessen Ausgabe als ASCII-Zeichen auf dem Bildschirm erscheint.

Zu jeder ROCK Funktion gibt es eine Online-Help-Funktion, die die benötigten Parameter anzeigt.

ROCK unterscheidet zwischen Fehlern und Warnungen; Fehler führen normalerweise zur Unterbrechung von Funktionen, sind aber häufig maskierbar. Die von den einzelnen Funktionen vorzunehmenden Fehlerprüfungen und -reaktionen sind recht präzise im Software-Functional-Specification-Manual beschrieben. Vom Entwickler selbst implementierte Funktionen können definierte Fehlerzustände auslösen (block), die etwa den Eintrag eines speziellen Wertes in einen Slot verhindern. Es besteht die Möglichkeit, den **ROCK Error Handler** durch eigene Fehlerbehandlungsfunktionen zu verändern. Dabei kann für jeden ROCK Error eine Funktion geschrieben werden, die den entsprechenden Fehler bearbeitet.

Die Hilfsmittel für die Erzeugung der Endbenutzerversion sind eingeschränkt, es besteht z.B. keine einfache Möglichkeit, deklarative Menühierarchien zu programmieren.

6.6. Dokumentation und Support

Zu ROCK gibt es zwei Handbücher, die miteinander sehr redundant sind. Aus der unklaren Verteilung der Information ergibt sich ein aufwendiges Suchen. Darüberhinaus ist der unzureichende Index in beiden Handbüchern ein fortwährendes Ärgernis, und es ist schwer, anhand des Index Informationen zu einem Thema zu finden.

Zur Qualität des Support von ROCK kann keine Aussage getroffen werden, da er während der Testphase nicht in Anspruch genommen wurde.

6.7. Zusammenfassende Bewertung

Die Repräsentations- und Manipulationsmöglichkeiten von Framehierarchien in ROCK sind sehr mächtig und erfüllen hohe Anforderungen. Im Vergleich zu Lisp-Systemen ist der Zwang zur Typdeklaration störend; er ist aber im Interesse einer schnelleren Abarbeitung einsichtig.

Durch die IFB und die Möglichkeit der Kontexte erscheint es realisierbar, eine Hierarchie von Wissensbasen aufzubauen.

Zur Systemsicherheit ist folgendes zu bemerken: Während der Verarbeitung von Objekthierarchien – im Gegensatz zum Laden von Objekt-Definition-Files – war kein Absturz des Systems zu verzeichnen. Das System ließ sich aber nach einer Arbeitsphase nicht zurücksetzen, vielmehr war ein Neustart erforderlich. Die Qualität von Fehlermeldungen in ROCK ist gut; beim Kompilieren werden die normalen Fehlermeldungen des Compilers ausgegeben.

Das Fehlen einer Regelkomponente sowie einiger Tools zur Unterstützung der Interaktion zwischen System und Entwickler (z.B. graphische Aufbereitung der Framehierarchie) sind die entscheidenden Schwächen von ROCK. ROCK allein ist in der derzeitigen Version sicher nicht geeignet, um ein komplexes Expertensystem mit aufwendigen Wissensverarbeitungsprozessen zu realisieren. Das Werkzeug stellt nur einen Teil einer Entwicklungsumgebung für Expertensysteme zur Verfügung, so daß wesentliche weitere Teile für eine vollständige Anwendung vom Entwickler selbst implementiert werden müssen.

7. Vergleich der Werkzeuge

Wissensrepräsentation

babylon: Frame-Taxonomie mit Vererbungsmechanismen • Behaviors • einfaches, nicht entwicklerdefinierbares Relationenkonzept • Regelkalkül • Constraints • Tasks zur strukturierten Ablaufsteuerung

KEE: Objekt-Hierarchien • eingeschränkt entwicklerdefinierbare Vererbungs–mechanismen • sehr einfaches Relationenkonzept • eingeschränkte Verwaltung von multiple worlds • Repräsentation der Regeln als Objekte • Realisierung eines einfachen ATMS

Knowledge Craft: Objekt-Taxonomie mit entwicklerdefinierbaren Vererbungsmechanismen • entwicklerdefinierbare Relationen zur Erstellung komplexer semantischer Netze • komfortable Verwaltung von multiple worlds

ProKappa: Objekt-Hierarchien mit verschiedenen vorgegebenen multiplen Vererbungsstrategien • entwicklerdefinierbares Relationenkonzept • Repräsentation von Regeln nur mit ProTalk

ROCK: Objekt-Taxonomie mit erweiterten entwicklerdefinierbaren Vererbungs–mechanismen • umfassendes benutzerdefinierbares Relationenkonzept zur Erstellung komplexer semantischer Netze • multiple worlds

Wissensverarbeitung

babylon: Forward-/Backward-Chaining • sequentielle Abarbeitung von Regeln • Prioritätenangabe für Regelabarbeitung nur im Regeltext möglich • Constraintpropagierung über bis zu 9 verschiedenen Lösungsstrategien • active values über Behaviors programmierbar

KEE: Objektorientierte Programmierung • active values frei programmierbar • erweiterbare und konfigurierbare Inferenzstrategien • Steuerung der Regeln in Common Lisp programmierbar

Knowledge Craft: objektorientierte Programmierung • active values frei programmierbar • Agendamechanismus zur Prozeßsimulation • Verarbeitung verschiedener Zeitrepräsentationen • regelbasierte Problemlösungsstrategien (OPS5 und PROLOG) im Objektsystem integriert

ProKappa: Objektorientierte Programmierung • C-Code frei integrierbar • keine benutzerdefinierbaren Inferenzstrategien • Regelmenge kann mit Hilfe von

	C bzw. ProTalk berechnet und der Inferenzmaschine übergeben werden • dynamisches Erzeugen bzw. Löschen von Objekten auf beiden Regelseiten möglich
ROCK:	Objektorientierte Programmierung in C^{++} • Integrated Frame Base (IFB) erlaubt Zugriff auf externe Objekte • active values frei programmierbar • Steuerung der Vererbungsstrategie durch Funktionen • kein Regelinterpreter

Visualisierung

babylon:	Graphische Entwicklungsoberfläche mit verschiedenen Browsern und Editoren • Unterstützung bei Erstellen einer windowbasierten Endbenutzeroberfläche basierend auf einer einfachen TellAndAsk-Schnittstelle
KEE:	Graphische Entwicklungsoberfläche für Regeln, Objekte und Taxonomien • Browser • graphische Inferenzdarstellung • Unterstützung für das Erstellen einer windowbasierten graphischen Benutzeroberfläche durch z.B. KEEpictures
Knowledge Craft:	Graphische Entwicklungsoberfläche für Regeln, Objekte und Taxonomien • Browser • Unterstützung beim Erstellen einer windowbasierten graphischen Benutzeroberfläche • Berechnung und Ausgabe von Statistiken
ProKappa:	Graphische Entwicklungsoberfläche für Objekte und Taxonomien • Browser • graphische Inferenzdarstellung • Unterstützung für das Erstellen einer windowbasierten graphischen Benutzeroberfläche
ROCK:	Keine eigene graphische Entwicklungsoberfläche und Browser • keine Unterstützung für das Erstellen einer Benutzeroberfläche • rudimentäre Darstellung von Frame-Taxonomien

Besonderheiten

babylon: *babylon* wird durch eine bindende Deklarationsreihenfolge und den begrenzten Zugriff auf die Basissprache Allegro Common Lisp die modulare Implementation von Wissensbasen erschwert.

KEE: In *KEE* ist ein paralleles Handhaben mehrerer Wissensbasen, die Regelhierarchisierung sowie die Trennung von Regeln und Inferenzsteuerung möglich. Eine individuelle Konfiguration der Systemkomponenten ist durch Basissprache Allegro Common Lisp sehr eingeschränkt.

Knowledge Craft: Der modulare Aufbau von *Knowledge Craft* ermöglicht eine individuelle Konfiguration von Komponenten einerseits für verschiedene Problemlösungsstrategien und andererseits zur Entwicklung von Benutzerschnittstellen über der gemeinsamen Repräsentationskomponente, welche die Basissprache Lucid Common Lisp vollständig integriert.

ProKappa: In *ProKappa* ist eine Konsultation mehrerer Wissensbasen sowohl vom Benutzer interaktiv oder per default zur Laufzeit möglich. Der Einfluß auf die Inferenzmechanismen bei der Regelverarbeitung ist nur eingeschränkt möglich.

ROCK: Das Fehlen einer Inferenzmaschine sowie einer graphischen Entwicklungsumgebung bei *ROCK* erweist sich als schwerwiegender Nachteil für die Realisierung komplexer Wissensbestände (insbesondere von heuristischem Wissen).

8. Zusammenfassung

In einem Projekt wie HYPERCON wo Umfang und Komplexität der Wissensdomäne mächtige Ausdrucksmittel erfordern, ist eine akzeptable Effizienz und Transparenz nur erreichbar, wenn darüber hinaus die Fokussierung unterschiedlicher Inhaltsbereiche und Detaillierungsebenen komfortabel unterstützt wird. Dafür werden Möglichkeiten der Modularisierung von Wissensbasen und des Inferierens in mehreren Welten benötigt. Aus dieser Sicht sprechen für:

babylon:	- Constraints
KEE:	- Komfort des Regelsystems
ProKappa:	- Komfort des Regelsystems
	- Qualität des Framesystems (Relationen)
ROCK:	- Qualität des Framesystems (Relationen)
	- Multiple Worlds
Knowledge Craft:	- Qualität des Framesystems (Relationen)
	- Multiple Worlds
	- Modulare Toolarchitektur
	- Zeitrepräsentation

Für das Projekt HYPERCON wurde wegen der Modularisierungsmöglichkeiten und des Komforts der multiple worlds trotz seiner leicht eingeschränkten Regelkomponente Knowledge Craft ausgewählt. Seine modulare Systemarchitektur ermöglicht eine individuelle Konfiguration von Komponenten für verschiedene Problemlösungsstrategien und zur Entwicklung von Benutzerschnittstellen und dadurch eine Skalierbarkeit in Abhängigkeit vom Ausbau des Systems.

Unser Dank für die engagierte und kooperative Unterstützung bei der Erstellung dieses Reports gehört Herrn Prof. Dr. Ipke Wachsmuth sowie unseren studentischen Mitarbeitern Jens Hamann und Jens Stoye.

März 1993

9 . Literatur

babylon

Christaller, Thomas (Hrsg.) — Die KI-Werkbank - Babylon: eine offene und portable Entwicklungsumgebung für Expertensysteme. Addison-Wesley. Bonn, 1989.

Referenzhandbuch — babylon, Softwarewerkzeuge für industrielle Anwendungen, Version 3.0, VW-GEDAS GmbH: Berlin, 1991.

Benutzerhandbuch — babylon, Softwarewerkzeuge für industrielle Anwendungen, Version 3.0, VW-GEDAS GmbH: Berlin, 1991.

Allegro Common Lisp — *User's Guide Vol. 1*, Version 4.1, Franz Inc. Document Number: D-U-00-000-01-20305-0-2-v1, März 1992.

Allegro Common Lisp — *User's Guide Vol. 2*, Version 4.1, Franz Inc. Document Number: D-U-00-000-01-20305-0-2-v2, März 1992.

KEE

Guy, L., Steele, J.R. — *Common LISP: The Language*, Digital Press, New Jersey, 1984. (ISBN 0-932376-41-X)

Wolf, S., Setzer, R. — *Wissensverarbeitung mit KEE*, Oldenbourg Verlag, München, 1991. (ISBN 3-486-21407-1)

Users' Guide — *Technical Manuals Vol. 1, :*KEE Version 3.1, IntelliCorp, Inc. Publication Number: K3.1-UG1, Mai 1988.

Interface Reference Manual — *Technical Manuals Vol. 2*, KEE Version 3.1, IntelliCorp, Publication Number: K3.1-IRM-1, 1987.

Core Reference Manual — *Technical Manuals Vol. 2*, KEE Version 3.1, IntelliCorp, Publication Number: K3.1-CRM-2, 1989.

TellAndAsk Reference Manual — *Technical Manuals Vol. 2*, KEE Version 3.1, IntelliCorp, Inc. Document Number: 3.1-TAA-2, 1989.

Rule System Reference Manual — *Technical Manuals Vol. 3*, KEE Version 3.1, IntelliCorp, Inc. Document Number: K3.1-RS3-2, 1989.

Rule Compiler Reference Manual — *Technical Manuals Vol. 3*, KEE Version 3.1, IntelliCorp, Inc. Document Number: K3.1-RC-4, 1989.

KEEworlds Reference Manual — *Technical Manuals Vol. 3*, KEE Version 3.1, IntelliCorp, Publication Number: K3.1-KW-3, 1989.

ActiveImages3 Reference Manual — *Technical Manuals Vol. 3*, KEE Version 3.0, IntelliCorp, Inc. Document Number: 3.0-R-A3, November 1986.

KEEpictures Reference Manual — *Technical Manuals Vol. 3*, KEE Version 3.1, IntelliCorp, Publication Number: K3.1-KP-2, 1989.

Common Windows Manual — *Technical Manuals Vol. 3*, IntelliCorp, Publication Number: CWM-2, 1989.

KEE 4.0 Release Notes — KEE Version 4.0, IntelliCorp, Inc. Publication Number: K4.0-RN-UNIX-1, Februar 1991.

Using KEE 4.0 on a UNIX Workstation — KEE Version 4.0, IntelliCorp, Inc. Publication Number: K4.0-UK-UNIX-1, Februar 1991.

KEEtutor — *KEEtutor: A Basic Course* (Module 1-12), IntelliCorp, Publication Number: KT-Mods1&2-Sun-3, 1989.

System Indices — *Technical Manuals Vol. 1,* KEE Version 3.1, IntelliCorp, Publication Number: K3.1-SI-1, 1990.

Knowledge Craft

Guy, L., Steele, J.R. — *Common LISP: The Language,* Digital Press, New Jersey, 1984. (ISBN 0-932376-41-X).

Guide to Documentation — *Guide to Technical Manuals,* KC Version 4.0/4.1, Carnegie Group, Inc., Pittsburgh, Juni 1990.

Reference Manual — *Manual,* KC Version 4.0, Carnegie Group, Inc., Pittsburgh, November 1989.

CRL — *Languages, Technical Manuals Vol. 1,* KC Version 4.2, Carnegie Group, Inc., Pittsburgh, November 1991.

CRL-OPS — *Languages, Technical Manuals Vol. 1,* KC Version 4.2, Carnegie Group, Inc., Pittsburgh, November 1991.

CRL-Prolog — *Languages, Technical Manuals Vol. 1,* KC Version 4.2, Carnegie Group, Inc., Pittsburgh, November 1991.

Workcenters — *Technical Manuals Vol. 2,* KC Version 4.1, Carnegie Group, Inc., Pittsburgh, April 1991.

Graphics — *Technical Manuals Vol. 3,* KC Version 4.1 Carnegie Group, Inc., Pittsburgh, März 1991.

Rapid Prototyping Methodology — *Tutorial,* KC Version 4.0/4.1, Carnegie Group, Inc., Pittsburgh, Juni 1991.

Timepak — *Technical Manual,* KC Version 4.2, Carnegie Group, Inc., Pittsburgh, November 1991.

Simpak — *Technical Manual*, KC Version 4.2, Carnegie Group, Inc., Pittsburgh, November 1991.

Statpak — *Technical Manual*, KC Version 4.2, Carnegie Group, Inc., Pittsburgh, November 1991.

ProKappa

Users' Guide — ProKappa User's Guide Version 2.0, IntelliCorp, Inc. Publication No: PK2.0-UG-2, Oktober 1991.

Programmer's Reference Manual — *Reference Manual Vol. 1*, Version 2.0, IntelliCorp, Inc. Publication No: PK2.0-RM1-1, Juni 1991.

C Functions Reference Manual — *Reference Manual Vol. 1*, Version 2.0, IntelliCorp, Inc. Publication No: PK2.0-MP-1, Juni 1991.

Developer's User Interface RM — *Reference Manual Vol. 2*, Version 2.0, IntelliCorp, Inc. Publication No: PK2.0-RM2-1, Juni 1991.

ProTalk Reference Manual — *Reference Manual Vol. 2*, Version 2.0, IntelliCorp, Inc. Publication No: PK2.0-CP-1, Juni 1991.

Interface Toolkits Manual — *Reference Manual Vol. 2*, Version 2.0, IntelliCorp, Inc. Publication No: PK2.0-IT-1, Juni 1991.

Data Access System Manual — *Reference Manual Vol. 2*, Version 2.0 DRAFT, IntelliCorp, Inc. Publication No: PK2.0-DA-1, Juni 1991.

ROCK

Software Functional Specification IMKA, Phase 1, Knowledge Representation, Draft Version 2.1, Carnegie Group, Inc., Digital Equipment Corporation, Ford Motor Company, Texas Instruments Incorporated, U S West, Inc., Juli 1990.

Software Functional Specification ROCK Version 2.1, Carnegie Group, Inc., Digital Equipment Corporation, Ford Motor Company, Texas Instruments Incorporated, U S West, Inc., Oktober 1991.

Knowledge Representation *Application Developer's Manual*, Part I, ROCK Version 2.1, Carnegie Group, Inc., Digital Equipment Corporation, Ford Motor Company, Texas Instruments Incorporated, U S West, Inc., Oktober 1991.

Data Management *Application Developer's Manual*, Part II, ROCK Version 2.1, Carnegie Group, Inc., Digital Equipment Corporation, Ford Motor Company, Texas Instruments Incorporated, U S West, Inc., Oktober 1991.